University of Aarhus
Department of Computer Science

Ph.D Dissertation

Static Analysis for JavaScript

Simon Holm Jensen

Supervisor:
Anders Møller

Submitted: January 28, 2013

Abstract

Web applications present unique challenges to designers of static analysis tools. One of these challenges is the language JavaScript used for client side scripting in the browser. JavaScript is a complex language with many pitfalls and poor tool support compared to other languages. This dissertation describes the design and implementation of a static analysis for JavaScript that can assist programmers in finding bugs in code during development.

We describe the design of a static analysis tool for JavaScript, built using the monotone framework. This analysis infers detailed type information about programs. This information can be used to detect bugs such as null pointer dereferences and unintended type coercions. The analysis is sound, enabling it to prove the absence of certain program errors.

JavaScript is usually run within the context of the browser and the DOM API. The major challenges in supporting this environment is to model the event loop of the browser and Document Object Model used to interface and modify the HTML displayed in the browser. We address both of these challenges in the design of our analysis.

Dynamic code evaluation is widely used in JavaScript applications. To accommodate this in the analysis, we add the Unevalizer component which can transform code on the fly to eliminate dynamic code evaluation. By studying the use of dynamic code evaluation in the wild, we have identified several common patterns. Many of these patterns can automatically be transformed into equivalent code without dynamic code evaluation and can then be analyzed further.

Acceptable performance is needed to make an analysis tool useful in practice. To that end we have designed an extension to the analysis called lazy propagation. Lazy propagation improves performance of the analysis by reducing the information that the analysis must consider in the program. Experimental validation of lazy propagation indicates a significant performance improvement.

The design of the analysis has been evaluated on a large selection of benchmarks taken from online sources. The results shows that the analysis is able to identify bugs in real code in reasonable time.

Resume

Web applikationer indeholder mange unikke udfordringer for designere af statisk analyse værktøjer. En af disse udfordringer er programmeringssproget JavaScript som bliver brugt til programmering i browseren. JavaScript er et kompliceret sprog med mange faldgruber og i forhold til andre sprog mangler det gode værktøjer til at hjælpe programmøren. Denne afhandling beskriver design og implementation af en statisk analyse for JavaScript.

Vi beskriver designet af et statisk analyse værktøj for JavaScript som er bygget ved brug af det monotone framework. Analysen infererer detaljeret typeinformation om programmer. Denne information kan bruges til at finde fejl i koden så som null pointer fejl og utilsigtede type konverteringer Analysen er sund, så den kan bevise programmer fejlfri for visse klasser af fejl.

JavaScript programmer bliver som regel kørt i en browser og bruger DOM APIet. De store udfordringer involveret i at understøtte dette miljø er browserens event loop og den objekt model som bliver brugt til at tilgå HTML siden. Vi adresserer begge disse udfordringer i designet af vores analyse.

Dynamisk kode evaluering er udbredt i JavaScript applikationer. For at kunne håndtere dette i vores analyse har vi udviklet Unevalizer komponenten som kan transformere kode med dynamisk kode evaluering til ækvivalent kode uden. Via et studie af brugen af dynamisk kode evaluering i rigtige programmer har vi identificeret flere gennemgående mønstre. Mange af disse mønstre kan automatisk transformeres til ækvivalent kode uden dynamisk kode evaluering og kan derved analyseres videre.

Acceptabl udførselstid for analysen er nødvendig for at den er anvendeligt i praksis. For at opnå dette har vi designet en udvidelse til analysen kaldet lazy propagation. Lazy propagation forbedrer udførselstiden ved at reducere mængden af information, analysen skal behandle i programmet. Eksperimentelle resultater viser betydelige forbedringer af udførselstiden ved brug af lazy propagation.

Designet af analysen er blevet evalueret på et stort udvalg af benchmarks fundet på Internettet. Resultaterne viser, at analysen er i stand til at finde fejl i rigtige programmer med et rimelig tidsforbrug.

Acknowledgments

I am indebted to my advisor Anders Møller for being a capable and useful mentor. He has been a great advisor both in sickness and in health during my time as a Ph.D student.

I thank the entire Programming Languages group at Aarhus University for creating a great working environment. I still do not understand what is so great about Foosball though.

A special thanks goes to my office mate Ian Zerny for having a decent taste in music and for not judging me on the days where I did not show up until after lunch.

I also thank Mathias Schwarz for a giving a meticulous review of this dissertation, which has greatly improved it.

I am also indebted to both Frank Tip and Satish Chandra who where both excellent hosts when I visited IBM Research Watson in Hawthorne and Bangalore respectively.

Finally I would like to thank my mother for supporting and encouraging me.

Simon Holm Jensen
Aarhus, January 27, 2013

Contents

Part I

Overview

Chapter 1

Introduction

Software development has always been a complicated endeavor with many pitfalls. History is full of examples where a simple software bug has had great, both financial and human, consequences. Several tools and techniques are used to mitigate bugs and help programmers write better code. Debuggers, testers, analyses, and type systems are all examples of techniques that can aid a programmer to better understand code and to find bugs.

The focus of this dissertation is on browser based *web applications*, which for our purposes is defined as programs running in a browser, communicating with a server, and written in JavaScript. Web applications have grown from modest roots to full size applications today. This means that programmers of web applications needs tools that can assist in program understanding and bug detection. Integrated Development Environments (IDE) usually provide facilities that aid in program understanding by for instance providing class diagrams and code completion. Bug detection, such as identifying possible null pointer derefences or uninitialized variables are provided both by IDEs and standalone programs such as Lint [53] or FindBugs [46]. Bug detection can also be extended to detect conditions that are not actual errors but still undesirable, such as unused variables or dead code.

1.1 Hypothesis

The central hypothesis of this dissertation is that we can design static analysis tools that give precise enough information about real programs to perform bug detection. Furthermore, to be useful the performance of the tool must be sufficient for practical use.

There are two major research challenges involved in this. The first one is the complexities of JavaScript itself. Type coercions, prototype chains and dynamic evaluation are all complicating aspects that we must take into account when building a static analysis for JavaScript. To be successful, an analysis must handle all these aspects while still being both reasonably performant and precise. The second research challenge is the browser environment where JavaScript code is executed. The two major factors in the browser is the event model and the Document Object Model which is an API that allows JavaScript to manipulate the HTML displayed to the user.

We test the hypothesis by designing and evaluating a static analysis tool for JavaScript. We describe how to handle the challenges of the very dynamic nature of the language and the execution model of the browser environment.

1.2 Method

This section describes the method that was used to investigate the hypothesis. Most of the work has been carried out in an iterative fashion. A set of initial assumptions were made by carefully reading the ECMAScript specification [23] and associated documentation. Based on these assumptions an analysis tool was implemented. The tool was evaluated on a number of standalone benchmarks collected online. The result of the evaluation was used to determine the weak points of the technique, the tool was then redesigned to overcome the identified weak points, and the next iteration was started.

1.2.1 Implementation

Implementing a tool for JavaScript means dealing with three distinct parts: (1) The core JavaScript language itself as defined in its specification. (2) The standard library also defined in the specification. (3) The event model and the DOM API added by the browser.

The authority on JavaScript is the specification published by ECMA [23], this specification has been the basis for implementing (1) and (2) above. Few ad-hoc extra features that are not in the specification are supported by browsers and used by programmers. This means that to handle real programs the design of an analysis must also support language features and APIs not part of any specification. In the work that is presented here non-standard features are implemented as needed by the chosen benchmarks.

The third part, the browser model and DOM, is defined by the W3C in a set of standards. It is not uncommon for browsers to deviate significantly from these standards so with the regards to the DOM, it is even more important to decide when to follow the standard and when to follow accepted practices. For the DOM we take the same approach as above, deviate from the standard when benchmarks require it.

JavaScript and its associated technologies are highly complex and sometimes interact in unforeseen ways. As JavaScript has no formal semantics there is not way for us to prove that the analysis is sound or the implementation bug free. We compensate for this by extensive testing, both unit testing and larger benchmarks. When a test is first run the output is carefully examined to determine if it is correct. If correct the test is added to the framework and the verified output is used to test later iterations of the tool.

1.2.2 Experimental evaluation

We validate each iteration of the tool on a set of benchmarks. We evaluate both precision and performance. The main performance metric used is execution speed, either real CPU time or the number of iterations needed to reach a fixpoint in the analysis. Number of iterations is the most robust measure as it is not affected by changes in the surrounding environment whereas CPU time is only comparable if benchmarks are run on the same hardware.

Evaluating precision correctly is more challenging. The overall question we want to answer is if the results yielded are useful for bug detection and program understanding. The only benchmarks we have available are programs that are released by the developers for other people to use. This means that they most likely have considerably fewer bugs than software under development has. Bug detection tools are primarily used during development so this means that simply counting the number of bugs detected on release quality software is not an adequate way to evaluate a tool. What we do instead is to measure the number of operations where the analysis does not yield a warning. This is done for each category of bugs. For instance, if the tool detects null dereference, we would measure the ratio of all dereference operations in the program to the number of operations where the analysis proves no null dereference can take place.

In addition we have also evaluated the information from the analysis with regards to program understanding. We measure two aspects of the analysis result that we believe are relevant to program understanding and comprehension: Call graph precision and the precision of the types inferred by the analysis. We measure call graph precision by calculating the ratio of call 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. To measure the precision of the types, we look at each read operation in the analyzed program. The analysis will determine the different possible types of values that can result from this operation. The fewer possible types, the more precise is the result. We calculate the average over all read operations in the program for an aggregated measure.

1.3 Structure

This dissertation consists of two parts. Part I gives an overview of the work done in the course of the author's Ph.D work. All steps outlined in the previous section are discussed. Part II contains the papers published as part of the work.

Chapter 2 starts with an introduction to web programming with the JavaScript language, including the Document Object Model, the browser event model, AJAX and the `eval` function for dynamic code evaluation. This chapter establishes the setting for the rest of the dissertation. Chapter 3 describes the static analysis techniques we use in the following chapters and describes the challenges that JavaScript poses to static analysis.

Chapter 4 presents TAJS, a static analysis for JavaScript that tackles many of the problems outlined in Chapter 3. TAJS has many facets and the presentation will describe the lattice structure used, lazy propagation which yields a significant performance gain by reducing the number of fixpoint iterations, specialized structures and transfer functions for supporting the DOM and the browser and the *Unevalizer* component that handles calls to `eval` in a sound manner. Chapter 5 describes the different experimental setups used to test the hypothesis. The results of the experiments are also discussed. Chapter 6 ends Part I with a conclusion on the work done.

The reader will benefit from familiarity with the basics of static analysis and its mathematical foundations. In particular the reader is expected to be

familiar with lattice theory. More background information can be found in textbooks such as [70] and [2].

1.4 Papers

The following papers were co-authored as part of the author's scientific work during his Ph.D studies at Aarhus University and are submitted along with this dissertation.

Type Analysis for JavaScript	with Anders Møller and Peter Thiemann. In Proc. 16th International Static Analysis Symposium (SAS), volume 5673 of LNCS. Springer-Verlag, August 2009.
Interprocedural analysis with lazy propagation	with Anders Møller, and Peter Thiemann. In Proc. 17th International Static Analysis Symposium (SAS), volume 6337 of LNCS. Springer-Verlag, September 2010.
Modeling the HTML DOM and browser API in static analysis of JavaScript web applications	with Magnus Madsen, and Anders Møller. In Proc. 8th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE), September 2011.
Remedying the eval that men do	with Peter A. Jonsson, and Anders Møller. In Proc. 21st International Symposium on Software Testing and Analysis (ISSTA), July 2012.

All papers can be found in Part II. The SAS 2010 paper appears in a tech report version while the rest are included as published with only layout changes.

The following paper was also coauthored during the Ph.D work, but is not part of the dissertation. The paper focuses on dynamic techniques instead of static analysis, which is the focus of this dissertation.

A framework for automated testing of JavaScript web applications	with Shay Artzi, Julian Dolby, Simon Holm Jensen, Anders Møller, and Frank Tip in In Proc. 33rd International Conference on Software Engineering (ICSE), May 2011.

Chapter 2

JavaScript and Web Development

The majority of web applications designed today consists of two distincts components: A client side running in the browser and a server side running on the server that the user connects to. The server side usually consists of a database and a program for accessing the database and generating HTML. The server side part of web applications can be written in almost any programming language, often used languages include, PHP, Java, Ruby and Python.

The client side part of a web application runs directly in the browser and the programmer therefore has less control over the environment in which his program executes. This constrains the choice of language to the what is available on potential user platforms. Currently the only language supported by all modern browsers is JavaScript. There are other languages supported through various plug-ins. However these are sandboxed and does not integrate well with the rest of the browser environment.

In this chapter we will briefly describe JavaScript and the DOM API. A reader already familiar with the language can skip this chapter.

2.1 ECMAScript and JavaScript

Strictly speaking, JavaScript as we know it is an implementation of *ECMAScript* which is standardized in ECMA-232 [23]. When one refers to a program written in JavaScript it usually means a program written in the ECMAScript language using the DOM API. ECMAScript 6[1], also called Harmony, is the culmination of several attempts to introduce a successor for JavaScript. No specification document has been released yet but some of the features planned for ECMAScript 6 already appear in some browsers. The work in this thesis does not address ECMAScript 6. However, the author is not aware of any ECMAScript 6 features that invalidate the work done in this dissertation.

JavaScript contains a number of features that makes it a challenge to analyze and detect bugs in:

- JavaScript is an object-based language that uses prototype objects to model inheritance. This is highly dynamic as prototype links can be manipulated at runtime.

[1] See `http://ejohn.org/blog/ecmascript-harmony/` for more information about ECMAScript 6.

- Objects are mappings from strings (property names) to values. In general, properties can be added and removed during execution and property names may be dynamically computed.
- Undefined results, such as accessing a non-existing property of an object, are represented by a particular value `undefined`, but there is a subtle distinction between an object that lacks a property and an object that has the property set to `undefined`.
- Values are freely converted from one type to another type with few exceptions. In fact, there are only a few cases where no automatic conversion applies: The values `null` and `undefined` cannot be converted to objects and only function values can be invoked as functions. Some of the automatic conversions are non-intuitive and programmers should be aware of them.
- Variables can be created by simple assignments without explicit declarations, but attempts to read absent variables result in runtime errors. JavaScript's `with` statement breaks ordinary lexical scoping rules, so even resolving variable names is a nontrivial task.
- With the `eval` function, a dynamically constructed string can be interpreted as a program fragment and executed in the current scope.

The popular name JavaScript and the syntax of the language relates to the Java and C languages, however this is a misconception. The semantics of JavaScript is closer to another prototype based language, Self [91] and the functional language Scheme [57]. Due to this misconception JavaScript is widely misunderstood and misused. This, combined with the forgiving nature of the language, leads to many faulty JavaScript programs being written. Browsers often silently mask these errors, so an unwitting programmer may write faulty JavaScript code without even realizing this.

2.1.1 Prototypes

Every JavaScript object has a *prototype* object. When looking up a property on an object for reading and it is it not found on the object the compliant interpreter must repeat the process for the prototype object. A given object's sequence of prototypes is the *prototype chain*.

The code below illustrates how JavaScript prototypes can be used to simulate class based inheritance

```
1  function Person(n) {
2    this.setName(n);
3    Person.prototype.count++;
4  }
5  Person.prototype.count = 0;
6  Person.prototype.setName = function(n) { this.name = n; }
7  function Student(n,s) {
8    this.b = Person;
9    this.b(n);
10   delete this.b;
11   this.studentid = s.toString();
12 }
13 Student.prototype = new Person;
```

The function `Person` is used to construct objects of type Person. Of course there is no part of JavaScript that enforces this type, it is merely a vehicle for understanding the code. All objects constructed using the `Person` function will have the prototype object `Person.prototype`. This means that all fields on the prototype will be shared between instances, making the `count` field behave like a static field and `setName` a member function. The `Student` function defines a sub-class of Person. The sub-class relationship is realized in two ways: (1) The function invokes `Person` as a super call and (2) all Person fields are available on Student objects. Property (1) is realized in lines 8 to 10 where the `Person` function is invoked in a way that ensures that the `this` identifier is bound to the new object being constructed. Property (2) is implemented by defining the `prototype` field of `Student` to be a Person object, thereby putting all fields defined by the `Person` function on the prototype chain.

2.2 The Document Object Model

When running in a browser, JavaScript programs can access and manipulate the HTML page that is presented to the user. This is done using the Document Object Model(DOM) API. The DOM is standardized by the W3C [60], however most implementations diverge from this standard in various ways. In this work, we have chosen to focus on the DOM as implemented and documented by the Mozilla project[2].

2.2.1 Events

The execution model for JavaScript running in a browsers is based around events. The user interacts with the program by triggering events on elements in the page. The programmer can register event handlers on objects. These handlers get invoked when the corresponding events are triggered. The DOM supports a multitude of different events: User triggered events such as an `onclick` event, AJAX events that are triggered when a call to the server returns and timeout events that are executed at timed intervals.

2.2.2 AJAX

Historically JavaScript and the DOM were exclusively used for client side scripting. With the advent of AJAX[3], it is possible for a program running client side to communicate directly with the server side part of the application. This is done asynchronously using a callback function so that a JavaScript application can interact with the server without blocking the browser.

AJAX allows for a much richer interaction between the client side and server side code and has paved the way for the more complex web applications we see on the Internet today. Prior to AJAX the only way for a JavaScript application to communicate with the server was to send regular HTTP request to the server, which also replaces the entire page.

[2] See `https://developer.mozilla.org/en/DOM`

[3] *Asynchronous JavaScript and XML.* The acronym is not accurate as both XML and JavaScript can be replaced by other technologies

2.2.3 An Example of DOM Usage and AJAX

```
1  <html>
2  <head>
3  <link rel="stylesheet" type="text/css"
4        href="style.css">
5  <script type="text/javascript"
6          src="ajax.js">
7  </script>
8  <script type="text/javascript">
9  var ajax = new AJAX();
10 var active = false;
11 var clicked = false;
12 var contentObj;
13 function mouseoverArticle() {
14   if (this==clicked) return;
15   if (active && active!=this) {
16     if (active==clicked)
17       active.className='selected';
18     else
19       active.className='';
20
21   }
22   this.className='over';
23   active = this;
24 }
25 function selectArticle() {
26   ajax.requestFile = this.id + '.html'
         ;
27   ajax.onCompletion =
28     function() {
29      contentObj.innerHTML = ajax.
            response;};
30   ajax.run();
31   if (clicked && clicked!=this)
32     clicked.className='';
33   this.className='selected';
34   clicked = this;
35 }
```

```
36 function init() {
37   var articles =
38     document.getElementById('articles')
39             .getElementsByTagName('li');
40   for (var i=0; i<articles.length; i++) {
41     articles[i].onmouseover =
           mouseoverArticle;
42     articles[i].onclick = selectArticle;
43   }
44   contentObj =
45     document.getElementById('content');
46 }
47 window.onload = init;
48 </script>
49 </head>
50
51 <body>
52 <div id="content">
53   <p>Click on one of the articles to the
         right.</p>
54 </div>
55 <div>
56   <ul id="articles">
57     <li id="article1">one</li>
58     <li id="article2">two</li>
59     <li id="article3">three</li>
60   </ul>
61 </div>
62 </body>
63 </html>
```

Figure 2.1: A simpel JavaScript program using DOM and AJAX.

Figure 2.1 shows an example of JavaScript that uses both DOM and HTML. It also demonstrates how JavaScript is often used, namely embedded in HTML documents. The programs first registers an event handler for the `load` event (line 47). This event handler (lines 36 – 46) in turn registers event handlers for `mouseover` and `click` events for each `li` element appearing in the element with ID `articles`. The `mouseover` events occur when the user hovers the mouse over the elements, causing the `className` to be modified thereby changing the CSS properties (lines 13 – 24). The `click` events occur when the user click on the elements, which causes the contents of the element with ID `content` to be replaced by the appropriate article being selected (lines 25 – 35). To save space, the associated CSS stylesheet and the file `ajax.js` that contains basic AJAX functionality is omitted.

2.3 Dynamic code evaluation

Like most scripting languages JavaScript supports dynamic code evaluation. Dynamic code evaluation is a language feature that can code load, parse and

execute code at runtime. This is most prominent in the `eval` function of JavaScript which takes as an argument a string that is executed in the currrent scope and environemnt. There exists other ways to do dynamic evaluation, including the `Function` object and certain DOM functions.

The `eval` function has historically been misused for things that could easier and more safely be done with other parts of JavaScript. A classical example of this is dynamic propety lookup. JavaScript supports dynamic lookup using the `[]` operator on objects. A programmer not aware of this might do dynamic property lookup using `eval` like follows:

```
1 var pp = eval("o." + p)
```

Where `p` is some dynamically computed property name. However this code is equivalent to the much simpler:

```
1 var pp = o[p]
```

Which also elimnates the risk of someone injecting code into the program by hypethetically manipulating the input to give `p` a malicious value, for instance:

```
1 var p = "foo;malicious_code()"
```

When passed to `eval`, this value of `p` behave as the programmer expects and looks up the property `foo`. However, it might also execute code that programmer likely did not intend.

`eval` also has legitimate uses: Some forms of meta-programming and macros can be implemented using `eval`. A quick way to parse JSON[19] is also to use `eval`. Most browsers support the function `JSON.parse` that parses only JSON code, which is a safer alternative to a full blown `eval` call which accomplishes the same thing.

Dynamic code loading is also used for application infrastructure. Modern web applications fetches code dynamically from the server using AJAX as needed, and executes it using `eval`.

2.4 JavaScript application frameworks

When writing JavaScript applications it is common to use one or more frameworks. These frameworks add functionality that makes development easier by providing tools for common tasks not provided by the language itself. These libraries often extend the environment in ways that is not possible in many other languages such as Java or C++. This extensibility is enabled by the fact that all of the build-in objects in JavaScript can be modified. One could for instance add a `escapeXML` method to the `String` object (which is the prototype of all string instances) making the method available on all string instances.

Many frameworks provide a mechanism for the programmer to program using classes. Which is often more familiar to developers coming from languages such as Java or C++. The prototype system is flexible enough to allow this kind of extension. Other commonly seen extensions are utilities for accessing the DOM tree in an easier fashion and other tools for building GUI applications.

Widely used frameworks include JQuery[4], Prototype[5] and Closure[6]. Frameworks are a challenges to handle precisely in an analysis. Our experiences with this is described in Section 5.2

[4] http://jquery.com
[5] http://http://prototypejs.org/
[6] https://developers.google.com/closure/

Chapter 3

Static Analysis Background

In this chapter we will describe relevant background information related to static analysis and JavaScript. We will build on this in the following chapters. The focus is on the monotone framework and the challenges JavaScript presents when the monotone framework is used to build an analysis for the language.

Section 3.1 discusses the runtime types of JavaScript, and how lattices can be used to model them in analysis. Section 3.2 describes control flow and how higher order functions means that data and control flow must be considered simultaneously. Section 3.3 outlines flow graphs that are used to represent programs. Functions are the key construct for structuring code and Section 3.4 discusses the challenges of interprocedural analysis. The chapter ends with a discussion of related work.

3.1 Values

A basic part data-flow analysis is to model the values that can appear at runtime. JavaScript is dynamically typed. This means that an analysis must be able to handle the fact that a given variable can contain values of different types, string and integer for instance, at different points during execution. JavaScript further complicates this by supporting a myriad of coercions between types.

The following JavaScript program illustrates a case where multiple types come into play for a single variable:

```
1 if (foo)
2     var x = "string"
3 else
4     var x = 42
5 o.p = x
```

In the above example the value of `x` depends on the boolean value `foo`. If the value of `foo` is not known, then `x` can be either a string or an integer when assigned to the property. For the analysis to be useful we must track both possibilities.

To model JavaScript values in the monotone framework we use lattices [21]. The lattices presented later in this dissertation are complex, and here we will just give a simplified example. Figure 3.1 shows two lattices: I for integers and S for strings. The lattice I tracks if a given integer value is negative, zero or positive. The value $\bot$ represents a value that is definitely not an integer, and

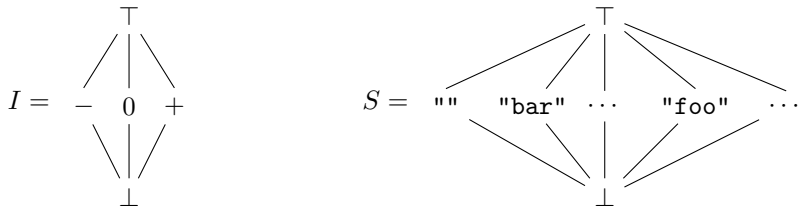


Figure 3.1: Example lattices for integers and strings.

$\top$ is an integer value that is unknown. The S lattice represents strings. Unlike I it contains infinitely many values, namely all possible constant strings. This is fine, however, as the lattice still has finite height.

A nice property of lattices is that they are closed under the cartesian product. Thus $I \times S$ is also a lattice. If we analyse the above code example using this lattice to model values, the value of `x` would be ("`string`", 42). Using the product lattice allows us to track both possible runtime values and thus not lose any information.

3.1.1 Objects

JavaScript has primitive values and object values. Objects cannot be represented in the same fashion as primitive values, so some other abstraction has to be used. A JavaScript program can potentially allocate an infinite number of objects, thus any abstraction will need to associate a given abstract object to multiple actual objects.

The allocation site abstraction [13, 54] is a common approach to this problem. It works by associating an abstract object to each program point in the program where objects may be allocated. The following code illustrates the technique:

```
1  var i = 0
2
3  function X() {
4    this.count = i++;
5  }
6
7  function f() {
8    return new X
9  }
10 f()
11 f()
```

For purpose of presentation we assume that each line in the code is a program point. This program has one allocation site of interest, l_8 on line 8 in function `f`. The function `f` gets invoked twice so at runtime two distinct objects are created, o_1 and o_2. On o_1 the `count` property has value 1, on o_2 it has value 2. Since both objects are allocated at l_8 they will be abstracted to the same abstract object. If we assume that we are using the integer lattice I from before, the abstract value of the `count` property on the abstract object would be $\top$. This example illustrates how precision is lost when we do abstraction.

An important concept with regards to objects is strong versus weak updates. When analyzing a property update on an object such as `o.p = 42` the abstract

object `o` potentially represents multiple concrete objects at runtime. Given this we can only do a weak update, meaning that the analysis must join `42` with the current abstract value of `p`. If we in some way can ensure that the abstract object only represents one concrete object, the analysis can do strong update where the current value is overwritten. We will return to this topic in Section 4.5.

3.2 Control flow

JavaScript has several features that complicate control flow analysis. Higher order functions mean that data-flow and control flow are entangled as functions can be passed as values. JavaScript also supports throwing and catching exceptions, which lead to alternative paths of execution that must also be tracked.

The following snippet illustrates code using higher order functions that an analysis must be able to handle.

```
1 function Cell(j) {
2     this.c = j;
3     this.get = function () {return this.c};
4     this.set = function (nc) {this.c = nc};
5 }
6 var cell1 = new Cell("string")
7 var cell2 = new Cell({})
8 console.log(cell1.get())
9 console.log(cell2.get())
```

This code defines a mutable cell containing one value. JavaScript has no notion of classes, but the function `Cell` acts as a class or a blueprint for Cell objects. Cell objects have two methods, `get` and `set` and one property storing the actual value.

To correctly analyze the above example the analysis must take the following into account:

1. The functions created on line 3 and 4 must be tracked to the `set` and `get` property of the two objects created on line 6 and 7. This is data-flow analysis.

2. When invoking the `get` method, the `this` identifier must be bound to the correct object in the body of the getter and setter function.

3. When analyzing the body of the `Cell` function, the `this` identifier must be bound to a newly created object, since it was invoked with the `new` keyword.

All of these features must be part of lattice and transfer functions for an analysis of JavaScript. This means that the lattice of values must also support functions and that the transfer function for an invocation must correctly setup the `this` identifier in the target environment.

This is an example of something that we will see again later, namely that to analyze JavaScript we must analyze all facets of the language in one analysis. As this example demonstrates, it is not possible to analyze control flow and data-flow separately.

3.2.1 Flow sensitivity

An aspect to take into account when designing a static analysis is flow sensitivity. If the analysis is flow sensitive, the ordering of statements in the program is taken into account during the analysis. A flow insensitive analysis will have one abstract state for the entire program whereas a flow sensitive will have one for each program point. Note that if context sensitivty is used then a flow insensitive algorithm will have one abstract state for each context. A flow sensitive one will have one abstract state per context at each program point. Context sensitiviy is discussed later.

Flow insensitive analysis has been applied to JavaScript in the WALA project [27] in the form of the classical Andersen style algorithm for pointer analysis with context sensitivity. In our work on JavaScript static analysis, we wish to do type analysis which means we must deal with a more sophisticated lattice than we would for a simpler points-to analysis. Our data-flow analysis will perform worse in terms of execution time than a flow insensitive analysis, but the result will in most real cases be more precise (see Chapter 5)

3.3 Representing programs

We need an appropriate representation of programs to carry out analysis. As we wish to do flow sensitive analysis, our representation must also reflect the order of the instructions. An often used approach is *flow graphs* where we model each function in the program as a graph. The source code of a program is then represented as a set of graphs, one for each function in the code. Since higher order functions are present we do not know statically which functions are invoked at a given call site and hence insertion of interprocedural edges are left to the analysis.

The nodes in the graph are basic blocks. Each basic block represents a sequential set of instructions in the original program with one exit point and one entry point. In Figure 3.2 is a small JavaScript fragment and the corresponding flow graph used in TAJS, the meanings of the individual instructions should be self-explanatory (for the specifics of the TAJS flow graph representation see Section 7.3). The figure shows how a call site has its own basic block since interprocedural edges will added during the analysis. Notice also that the if-statement on line 2 is compiled to a basic block with one successor for each branch.

Exception flow is represented by the gray arrows indicating where control should jump to if an exception is raised. The instructions in each basic block represent different operations in the program, such as invoking a function or adding numbers. Usually one statement in the program being represented corresponds to multiple instructions on the flow graph level.

3.4 Functions

When analyzing programs consisting of multiple functions one can do either inter- or intraprocedural analysis. When applying intraprocedural analysis, flow is not propagated across function boundaries and all information that can be extracted will be local to the current function. This is adequate for some

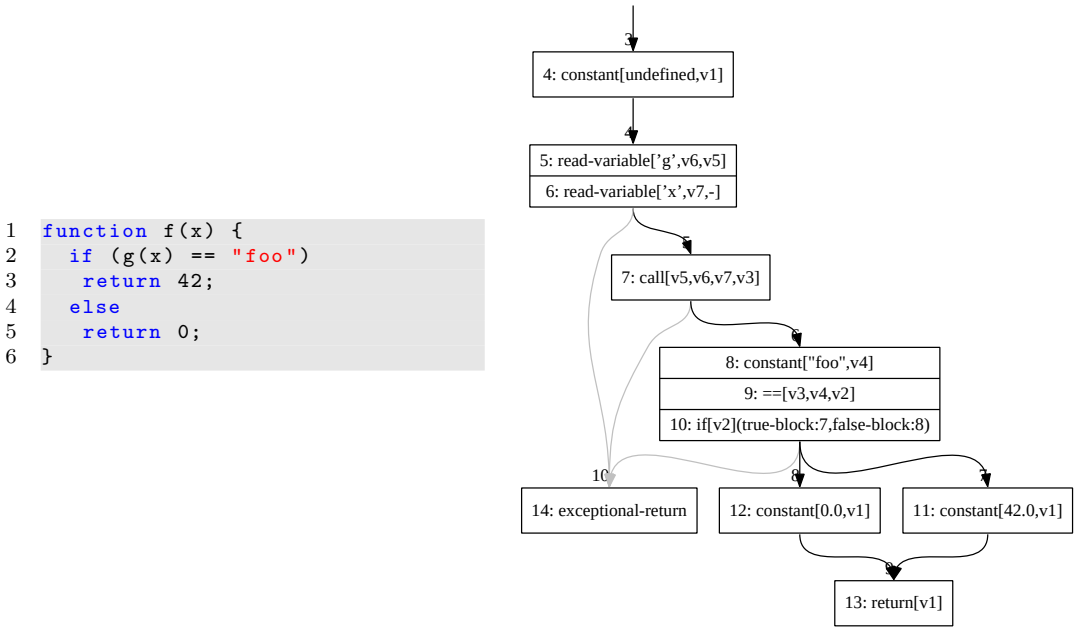


Figure 3.2: Code fragment and its corresponding flow graph.

analyses, such as ensuring that a variable is initialized through all paths of the program such an analysis is performed by many compilers. Intraprocedural analysis is, however, not enough for the kind of data-flow analysis we wish to perform in this work. If no information is propagated across function boundaries we would in essence have to assume that a given function inovcation can modify any part of the global state. Functions are such a big part of programming that assuming this would lead to an unacceptable lose of precision.

3.4.1 Interprocedural analysis

Propagating information across function invocations is interprocedural analysis. As mentioned above, in JavaScript control and data-flow are interdependent so in the general case we do not know which function a given invocation will call before the analysis. Special interprocedural edges are added during analysis as data-flow facts are discovered. For each discovered function a call edge and a return edge are added.

Naively adding interprocedural edges quickly leads to unacceptable loss of precision. The following small example demonstrates this:

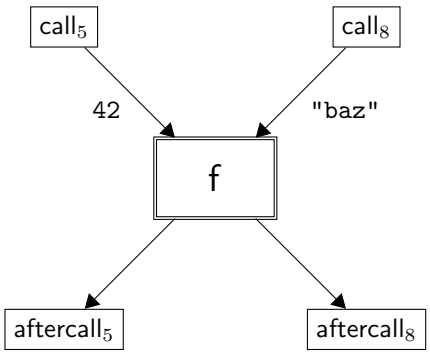


Figure 3.3: Interprocedural flow.

```
1  function f(x) {
2    return x;
3  }
4  function g() {
5    var foo = f(42)
6  }
7  function h() {
8    var bar = f("baz")
9  }
10 g()
11 h()
```

In this example `f` is the identity function, which gets invoked from two different call sites on line 5 and 8 with two different argument values. In a naive interprocedural analysis call and return edges would be added from both call sites to `f` yielding the call graph seen in Figure 3.3. The entry of `f` receives incoming flow from two sources, meaning that the abstract value of `x` would be $(+, \text{"}\texttt{string}\text{"})$ using our example lattice from before. The major problem here is unmatched call and return edges. In our example the abstract state at call_8 is propagated to $\mathsf{aftercall}_5$ which is an infeasible flow in the program. In bigger programs, unmatched call and return is a significant problem as completely unrelated parts of the program can interfere with each other.

3.4.2 Context sensitivity

To combat infeasible flow and the imprecision it leads to we add context sensitivity. When context sensitivity is enabled, a finite number of contexts exists in the analysis. Each function is now analyzed in a context and different flows in different contexts are kept apart.

A commonly used context set is call strings [85]. In this setting a context element is a string of functions in the program. To ensure that the set is finite the strings are bounded at some length n. In the previous example, the function `f` would be analyzed in two different contexts, namely the call strings of length one, `h` and `g`. This means that the two different arguments will be kept separate and the infeasible flow will not occur.

JavaScript offers many possible choices of contexts. As an object oriented language, many functions are often associated with objects. It therefore make sense to experiment with making objects part of the context [69]. As will be

discussed in Section 4.3 we have chosen the value of the `this` pointer at invocation as the context. In effect this defines the context as a set of allocation sites.

3.5 Computing the fixpoint

A fixpoint of a function is a value mapped to itself by the function. A fixpoint to the transfer functions we use to model the language will be an approximation of the runtime values.

A common way to compute the fixpoint of the transfer functions is using the work list algorithm. To simplify the presentation, we ignore contexts here.

```
1: for every flow graph node n: in(n) ← ⊥
2: insert every n into worklist W
3: while W not empty do
4:    n ← W.next()
5:    old ← in(n)
6:    new ← transfer(old, n)
7:    if old ≠ new then
8:       in(n) = new
9:       for all n′ ∈ succ(n) do
10:         in(n′) ← in(n′) ⊔ new
11:         W.insert(n′)
12:      end for
13:   end if
14: end while
```

Line 1-2 of the algorithm assigns every node in the flow graph an initial lattice value, namely the bottom value $\bot$ and inserts the node into the work list W. After this, the algorithm iterates until a fixpoint is reached and the work list is empty. Each iteration starts on line 4 by removing the next node from the work list. In lines 5-6 the old lattice value associated with the node is saved and the new value is computed using the appropriate transfer function. If the value changed by applying the transfer function we need to propagate this to the successors of the node (line 7). If there was a change we first store the new lattice value (line 8), and then update the lattice value of each successor using the join operation.

3.6 Alternatives to static analysis

Static analysis is but one approach that can derive the information needed for bug detection from JavaScript programs. This section discusses related work and the pros and cons compared to static analysis as used in this work. Work that is more closely related to this dissertation is discussed in Section 4.9.

3.6.1 Type systems

While JavaScript is an untyped language several attempts at adding a type system to it has been undertaken. Thiemann [89] presents a type system for JavaScript aimed specifically at analysis and detecting coercion errors. The

system is defined on subset of JavaScript missing some essential features of the language such as prototypes and exceptions. The system is proved sound but no typing algorithm is given.

Anderson et al. [3] present a type system for JS_0, a core JavaScript calculus, including a type inference algorithm. As before, this calculus does not support essential JavaScript features such as the prototype mechanism. The type system models the dynamic JavaScript objects where properties can be potentially absent or definitely present.

Type systems can also be used as a part of an analysis. Guha et al. [37] present a type system for JavaScript and discuss the challenges the `typeof` operator presents to typing. As a solution a flow sensitive analysis is used. The flow analysis inserts runtime checks of the programmer's `typeof` usages. The type system integrates with these runtime checks and allows common uses of `typeof` to be typed.

A common features of retro-fitted type systems for JavaScript is that they require user annotations of some kind in the code. Requiring this raises the bar of entry as extra effort is now required on part of the programmer and it might confuse other development tools such as IDE editors. Static analysis as presented here does not require any extra annotations of the code.

3.6.2 Dynamic approaches

Perhaps further removed from static analysis than type systems are approaches based on dynamic analysis. Dynamic analysis involves actually executing the program in some manner. This presents significantly different challenges than static analysis. In static analysis the interesting properties are soundness, precision and performance. While performance is of course still an issue, soundness and precision are not. The main metric of dynamic analysis is coverage. For the analysis to be useful, a large part of the code must be executed. To achieve this the analysis must in some manner discover inputs that enables it to cover a high number of execution paths.

In the Kudzu project [83] Saxena et al. use an instrumented WebKit engine to dynamically explore JavaScript web applications. They divide the input space into two spaces, the event space and the value space. The event space is the DOM events triggered and the value space is the value entered into the form elements on a page. The event space is explored by randomly generating event sequences and triggering them on the page. The value space is explored using concolic execution [84, 30] of the JavaScript code. The symbolic execution engine has an elaborate model for reasoning about string values and string operations. Due to the complexity of the approach the system needs a long time to run. The experimental evaluation operates with a 6 hour timeout.

The author of this dissertation and others present the Artemis project [4], which also does automated testing of JavaScript web applications. Artemis is based on random feedback directed testing [73] where the generation of random test input is guided by information collected about the execution. Artemis generates both random input events and random inputs for form elements. Several heuristics are used to gather information about the program during execution. For instance constant mining is used to generate inputs and the application is monitored to see which events it has listeners registered for.

Compared to Kudzu described above, Artemis is more lightweight, requiring only a few minutes to do a test.

Comparing dynamic approaches to static analysis is difficult. The dynamic approach will always be limited to testing, no guarantees about the program behavior can be given. On the other hand a dynamic tool will never issue a false warning, as all information is based on actual executions.

3.6.3 Semantics

As mentioned JavaScript lacks a formal semantics. This means that designing a sound analysis will always be a best effort enterprise, no proof of correctness can be given. Given a semantics and a formal definition of the analysis one can prove soundness of the analysis either manually or using an interactive theorem prover.

Maffeis et al. [64] present a small step operational semantics formalizing ECMAScript as laid out in the official specification. It is the first work to address the full language instead of various core languages based on JavaScript. The semantics reflect the complexity of JavaScript and the full account is over 40 pages. The authors use their semantics to design and prove soundness of a security analysis of JavaScript.

λ_{JS} is a core JavaScript language presented by Guha et al. [36]. What sets λ_{JS} apart from the other core languages used is that it includes a desugaring algorithm that takes programs in JavaScript and turns them into λ_{JS} programs. Later work[1] introduces Mechanized-λ_{JS} where soundness and presevation of the operational semantics of λ_{JS} are proved using the Coq proof assistant. Along with the desugaring mechanism this provids a starting point for defining static analyses whose soundness can be mechanically verified. Another semantics called S5 by Politz et al. [76] is aimed at ECMAScript 5. The work is based on λ_{JS} and focuses on modeling `eval` and the getter and setter functionality introduced by ECMAScript 5.

Horn and Might [43] demonstrate how a semantics can be used to build an analysis. Starting with λ_{JS} they derive an abstract machine for JavaScript using the refocusing approach of Danvy et al. [20]. Starting from this machine they use abstract interpretation to derive a sound analysis for JavaScript. As an example, the authors shows how the process can derive a k-CFA analysis for JavaScript.

[1] See the following blog post from the authors: `http://brownplt.github.com/2012/06/04/lambdajs-coq.html`

Chapter 4

TAJS

The work described in this thesis has been implemented in the static analysis tool, TAJS. TAJS is whole program, flow-sensitive analysis, supporting the entire ECMAScript language and parts of the DOM. TAJS also includes support for applications running in the browser event loop. TAJS models the semantics of JavaScript in a sound manner with a rich lattice structure. The results of TAJS have been used to find bugs and aid in program comprehension.

TAJS is open source and the code is freely avaliable at the following URL:

`http://brics.dk/TAJS`

4.1 Design choices

TAJS is at its core an application of the classic monotone framework [58], however within the bounds set by the framework a lot design choices still have to be made. In this section we will describe some of the choices, motivate why they were made and compare with alternatives.

4.1.1 Whole Program

TAJS is a whole program analysis. This means that the entire program must be available at analysis time, and that if part of the program changes, the whole program must be analyzed again. The obvious downside of this is performance. If, for example, the analysis were to be used interactively in an IDE the cost of re-analyzing the code every time it is changed could be prohibitively expensive. In this work we have chosen to focus in precision first, leading us to focus in the design of the lattice. In the Gatekeeper project Livshits and Guarnieri [33] design an incremental version of a points-to analysis for JavaScript. When new code is added to the application, the analysis information can be updated without re-analyzing the old code. While TAJS does computes points-to information, it also computes detailed typed information about the program which is hard to do in an incremental fashion. We have instead focused on other techniques to improve performance such as recency and lazy propagation.

Other whole-program analyses include ASTREE [8, 9] which is an analysis for C programs. ASTREE shares several design features with TAJS: ASTREE is a whole-program analysis, which is sound and designed primarily with precision in mind. ASTREE proves the absence of run-time errors in C programs,

TAJS has been applied to the same for JavaScript [52]. Since C is a significantly different language from JavaScript, the specific properties checked are very different.

4.1.2 Sound Approximation

TAJS is a sound modeling of the ECMAScript semantics. This means that if the program under analysis contains an error in the class detected by the tool, TAJS will find it. Conversely it also means that TAJS is able to prove the absence of errors in programs. Compare this with unsound tools such as FindBugs for Java [46], where the focus is on detecting common bugs without issuing too many false warnings. The downside of unsoundness is that no guarantees exists. The tool detecting no bugs is not a guarantee that the program is error free.

False positives are a negative effect of sticking to soundness. In TAJS we alleviate this by prioritizing the warnings issued. Definite errors that are guaranteed to occur at runtime are displayed before warnings that are potentially false positives.

4.2 Overview

The present chapter outlines the structure of TAJS. We will discuss the main components and techniques. Whenever appropriate we will refer to the papers in Part II where more details can be found.

As mentioned, TAJS is an instantiation of the monotone framework. The core algorithm is a fixpoint iteration over the program state lattice that is presented in Section 4.3. The semantics of JavaScript is modeled by a set of transfer functions over the lattice, presented in Section 4.4. To model the DOM and browser environments, the lattice of TAJS is extended to track event handlers and the DOM tree. This is presented in Section 4.7. Dynamic code evaluation such as the `eval` function requires special attention to model soundly. The fixpoint algorithm is augmented to replace calls to `eval` whenever possible in a sound manner, this is discussed in Section 4.8

TAJS includes several optimizations to improve performance: The recency abstraction boosts precision of object reads by ensuring strong updates in many cases. The recency abstraction is discussed in Section 4.5. Another technique we have developed is lazy propagation, which will be discussed in Section 4.6. Lazy propagation reduces the number of iterations needed to reach the fixpoint by only propagating values into a function if the function has previously been determined to access those values. Futhermore it reduces the size of an abstract state thereby reducing the memory consumption of the analysis.

4.3 Lattice

TAJS employs a very detailed lattice to model the possible runtime values of a JavaScript program. The current design of the lattice is the result of an experimentally driven process where each refinement is motivated by behavior observed on benchmark programs.

This section will present an overview of the lattice and highlight interesting portions. Full details are in the paper, Section 7.4.

4.3.1 Program state

TAJS is flow-sensitive so each program point has an associated abstract state. The lattice for TAJS is therefor a map from program points to abstract states:

$$\mathsf{AnalysisLattice} = V \times N \ \rightarrow \ \mathsf{State}$$

In this definition V denotes contexts (which will be described later) and N is the set of program points. TAJS represents program source code using flow graphs the exact nature of which is discussed in the paper Section .

The State lattice describes the state of the whole program. This includes two elements namely the stack and the store:

$$\mathsf{State} \ = \ (L \ \hookrightarrow \ \mathsf{Obj}) \ \times \ \mathsf{Stack} \ \times \ \mathcal{P}(L) \ \times \ \mathcal{P}(L)$$

The store is modeled as a partial function from the set of object labels, L, to elements of the abstract object lattice Obj. The lattice Stack is discussed below and the two remaining sets track object labels that are maybe/definitely summarized (see recency Section 4.5). Note that the store is modeled with a partial function as not all possible objects labels have necessarily been used at a given program point.

The Obj lattice models JavaScript objects created at runtime. It is defined below:

$$\mathsf{Obj} \ = \ (P \ \hookrightarrow \ \mathsf{Value} \ \times \ \mathsf{Absent} \ \times \ \mathsf{Attributes} \ \times \ \mathsf{Modified}) \ \times \ \mathcal{P}(\mathsf{ScopeChain})$$

Objects consists of a map from property names to values and flags indicating if the property is possibly absent or modified. Futhermore, the specification defines a number of features to be put on object properties, such as if the object is read-only for instance. The scope chain component is needed if the object is in fact a function.

The Stack lattice models the JavaScript runtime stack. In TAJS flow graphs complex expression are broken down to simple forms that read, and write so called temporaries. A temporary has no directly equivalent in the JavaScript source code, and is an artifact of the representation. The stack includes the values of all temporaries:

$$\mathsf{Stack} \ = \ (T \ \rightarrow \mathsf{Value}) \ \times \ \mathcal{P}(\mathsf{ExecutionContext}) \ \times \ \mathcal{P}(L)$$

Execution context is JavaScript terminology for stack frame and consists of all the currently active scopes (called the scope chain). The last element of Stack is the set of all object labels that are reachable from the stack which is used for abstract garbage collection [68]. More details about execution contexts and scope chains are in the paper, Section 7.4.

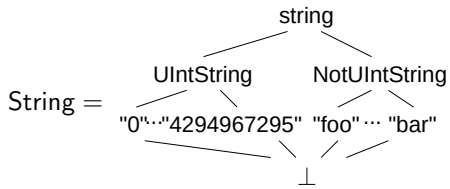
4.3.2 Abstract values

This section explains the model of JavaScript's runtime type system used by TAJS. The starting point is the product lattice Value which models all possible

values that can occur at runtime:

$$\mathsf{Value} \quad = \quad \mathsf{Undef} \times \mathsf{Null} \times \mathsf{Bool} \times \mathsf{Num} \times \mathsf{String} \times \mathcal{P}(L)$$

Like many programming languages JavaScript distinguishes between primitive types and object types (although the elaborate coercion and boxing mechanisms blurs this distinction somewhat). The first 5 lattices model the possible primitive values in JavaScript and the last the object types (objects are represented by allocation sites). All of the lattices are discussed in detail in the paper, Section 7.4. Here we will highlight just one lattice, String, which demonstrates how the design is motivated by real JavaScript code.



A value in the String lattice can be either a constant string, a string representing an unsigned integer or a string *not* representing an unsigned integer. The distinction between UIntString and NotUIntString is motivated by the fact that JavaScript supports dynamic property lookup on all objects: Take for example the below code snippet that initializes an object using a `for` loop:

```
1  var foo = {}
2  for (i = 0; i < 1; i++) {
3      foo[i] = 1;
4  }
```

The value of `i` is coerced to a string when doing the lookup on line 3. Since the value of `i` is not constant, a reasonable approximation might simply be the top lattice element, string, denoting any possible string (indeed this was the strategy used in earlier versions of TAJS). This however leads to complications: The only way to do a dynamic lookup with an unknown string soundly is to return *all* values that can be reached via a lookup on that object. If the lookup is a read operation this includs properties that are reachable through the prototype chain. In the example, which is a write, the effect is that the write on line 3 risks touching completely unrelated properties on the object.

However in TAJS the value of `i` coerced to a string would be UIntString, which limits the values returned by the lookup to values reached using arrays indices, thus ensuring that no unrelated values either on the object itself or on the prototype chain is touched.

4.4 Transfer functions

The TAJS lattice models the runtime values. To model the semantics of JavaScript we have defined a set of transfer functions defined on the lattice. As mentioned previously, soundness is a design criteria for TAJS, and therefore the transfer functions must faithfully model JavaScript as laid in the specification, including the numerous corner cases.

TAJS represents programs as flow graphs, and has a corresponding transfer function for each flow graph node. Section 7.3 includes a detailed listing of node types in TAJS.

Procedure calls are represented by call nodes. The analysis adds call edges to these nodes when the target functions are determined at run time. The semantics of a procedure call in JavaScript are surprisingly complicated, but the lattice of TAJS is rich enough to capture them: A function call at call node c involves a lattice value $f \in \mathsf{Value}$, a base object o and a (possible empty) list of arguments $v_1, \ldots, v_n$. During analysis TAJS takes the following steps to carry out the function call from c to f:

1. If the abstract value f is possibly something else than a function value, a warning is signaled.
2. Edges from c to the entry node of all functions in f are added to the flow graph. Likewise edges from the return nodes in these functions to the *after-call* node of c are added.
3. We must also handle exceptional flow. To accomplish this each function in the flow graph has an exceptional exit node. If c has an exception edge to c_{exn} then all exceptional exit nodes of functions in f will also have an edge to c_{exn}.
4. A new execution context is pushed onto the call stack.
5. The `this` identifier is bound to o in the new scope. o is also used to setup the scope chain in the new execution context.
6. The `arguments` array is populated with arguments v_i and added to execution context as well.

Furthermore TAJS includes transfer functions specifically for DOM related features. These are discussed in Section 4.7. TAJS also includes transfer functions for many of the built-in functions of JavaScript. This is a design decision, alternatively we could have implemented these functions in JavaScript and simply included them in the analysis. We chose not to do this out of performance considerations.

4.5 Recency abstraction

The notion of strong updates is important with regards to precision. Given an update node in the flow graph $\mathsf{write\text{-}property}[l, p, v]$ that writes the abstract value v to the property p of an object allocated at allocation site l, we wish to model this as precisely as possible while remaining sound. If we know that l only denotes one possible runtime object, then we can perform a strong update and overwrite the current value with v. In general more than one object can be allocated at a given allocation site and to remain sound we must settle for a weak update which joins v with current value of p on l.

In TAJS we use the recency abstraction, first presented by Balakrishnan and Reps [7] in the context of analyzing x86 executable and later applied to JavaScript by Heidegger and Thiemann [40]. With recency enabled, TAJS associates two object labels $l_@$ and l_* with an allocation site l. The $l_@$ set represents the most recently allocated object at site l. It is a singleton and therefore permits strong updates to be performed. l_* represents all other objects allocated at l.

A common JavaScript pattern for initializing an object is to first create an empty object, and then subsequently assign all properties to this empty object. Without recency each write to the empty object would have be a weak update,

meaning that the analysis would mark the property as being potentially absent. A later read to one of these properties would then trigger a false warning. Using recency we can do strong updates in this case while maintaining soundness.

4.6 Lazy Propagation

Experimental investigations of the behavior of TAJS on our benchmarks revealed that the analysis spends a significant amount of time doing work that does not help bringing it closer to a fixpoint. For instance the below one-line function from the benchmark `richards.js` was analyzed 18 times.

```
1  TaskControlBlock.prototype.markAsRunnable = function () {
2    this.state = this.state | STATE_RUNNABLE;
3  };
```

The cause of all this is an artifact of how TAJS did intraprocedural analysis: Assume that the above function is invoked from two call sites. If the abstract state at either of the call sites is changed, all successor blocks (both intra- and inter-procedural ones in the case of call sites) are added to the work list to be analyzed again. The abstract state contains the state of the entire program, and there is no guarantee that the change being propagated to `markAsRunnable` above is being accessed at all in the function.

Lazy propagation is an attempt to alleviate this problem, by delaying propagation of a given part of the state until it is actually needed. In this section we will present the technique using an example, Chapter 8 gives a more rigorous treatment. Note that lazy propagation is a technique that can be applied to any data-flow analysis, it is not specific to TAJS.

4.6.1 A call graph

For our example we use the call graph in Figure 4.1. The figure shows 4 functions, f_1, f_2, f_3 and f_4. The function f_1 writes to the location x (an object property) and later invokes the functions f_2 or f_3. Function f_2 does not touch x and just invokes function f_4 while f_3 writes a different value to x. Function f_4 then reads x. In the figure the directed lines represent control flow and the bullets represent elements of the State lattice. We assume that the analysis discovers all call edges prior to visiting *read*. This example ignores data-flow for function returns and calling contexts.

4.6.2 Analysis with lazy propagation

When the analysis enters f_1 the x property has neither been written or read, and is therefore $\bot$. After $write_1$ has been analyzed, x has the lattice value v_1 at the entry of $call_1$. After parameter passing, the state of the program after $call_1$, is stored in the two states, s_1 and s_2 located on the call edges. Note that in these two states x is still v_1.

Since neither f_2 or f_3 have been visited by the analysis yet, there is no definite information that x is actually referenced in these two functions, and therefore there is no need to propagate the actual value of x into the functions. Instead it is replaced by the placeholder value unknown at the entry. The unknown value is a special value introduced by the lazy propagation framework,

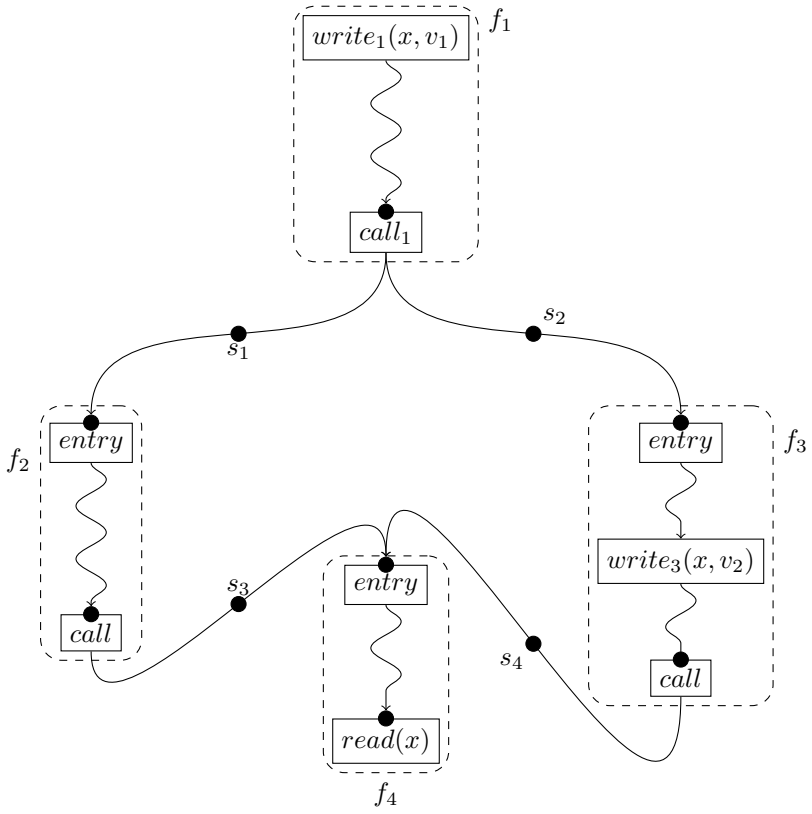


Figure 4.1: Flow graph of the example

and it represents an abstract value that the analysis has not yet propagated since it has not yet encountered any accesses of this value.

In f_2, x is neither written nor read, and therefore x is still unknown at s_3. If we assume that x is updated with strong update[1] at $write_3$ in f_3 it has value v_2 in s_4 after f_3 has been analyzed. As with f_2 and f_3, f_4 has not yet been visited and so at the entry state of f_4, x is set to unknown.

When the analysis tries to read the value of x at the read node, the presence of the unknown value indicates that the actual value has not yet been propagated to this program point. To proceed the analysis will have to recover the actual value.

The first phase of the recover operation constructs a directed graph G, where the nodes are functions from the original program. The set of nodes in G consists of all functions where x is unknown at the entry node. A subset of the nodes in the constructed graph are *roots*. A root is a function where at least one of the incoming edges has a known value for x on it. Note that roots can have incoming edges.

For the present example, G consists of the functions f_2 and f_4, where f_2 is connected to f_4. The set of roots in G is $\{f_2, f_4\}$ as these are exactly the

[1] if we assume weak update, a recover operation would be triggered at this point

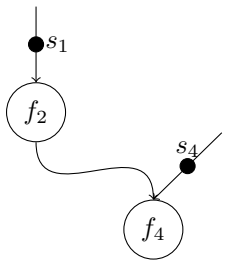


Figure 4.2: The recover graph G.

functions where the unknown value was introduced. Figure 4.2 illustrates G. The next step of the algorithm propagates the known value of x from the call graph states (where x has a known value) into the root nodes. After this, the value is propagated through G until a fixpoint is reached in the standard manner. When the fixpoint is reached, all function entries in G will have the actual value of x at the entry states and the analysis can proceed.

With lazy propagation only the parts of the state that are actually used inside a function are propagated into it. For JavaScript this often means a big reduction in the average size of the abstract states, as many of the objects in the standard library can be represented by unknown instead.

4.7 Modeling the Browser

Most JavaScript applications run in the context of web applications. To support this TAJS has been extended with three different functionalities: (1) A parser for HTML that creates an abstract model of the HTML page that contains the JavaScript, (2) transfer functions for the DOM functions that access the HTML page, and (3) a modeling of the event model of the browser.

The model of the HTML page needs to reflect both the elements defined statically by the actual page and the nodes created dynamically by the JavaScript code. As HTML nodes can be created dynamically, we need a bounded representation to ensure termination. In TAJS we chose to represent the HTML DOM model with on abstract object per node type, i.e. we have one abstract object representing all `HTMLInputElement` elements. The model exploits the prototyping mechanisms of JavaScript and places all common functions supported on another abstract object `HTMLInputElement.prototype`. This approach also mirrors the inheritance structure laid out in the DOM specification and is bounded.

It is common in DOM applications to map between element IDs (set with the `id` attribute in HTML) and DOM objects using `getElementByID` and related functions. To support this in TAJS we have extended the State lattice with a map from the ID names used in the HTML to the corresponding abstract object.

The DOM API itself is defined in the form of numerous transfer functions on the different abstract objects. Section 9.4 has more details on the scope of this implementation.

4.7.1 Event Model

To correctly analyze code using the DOM API we need to model the browser. The browser executes JavaScript in an event driven fashion. When code is loaded in the browser a portion of the code (called top level) is first executed. Then events are triggered by the surrounding environment (for instance AJAX and user events). If the application has registered a handler for a triggered event, that handler gets executed.

This new model complicates analysis and presents a challenge to precision since every event handler can conceptually be viewed as a possible entry point in the code. Modeling this in a sound way presents a trade-off between performance and precision (and code complexity in the analysis). Events have an ordering depending on when they were registered since event handlers can register and remove other event handlers. Properly modeling this order would require TAJS to build some form of state machine indicating which events have handlers at a given time.

Instead we take a simpler approach: The event loop is encoded at the flow graph level and the state lattice is expanded to track registered event handlers. To ensure soundness we separate the handlers into multiple sets depending on their type. Currently the following 6 categories are tracked:

load Event handlers that get triggered when the page is loaded.

mouse, keyboard The different kinds of events triggered by the user by interacting with the mouse and keyboard. We distinguish between mouse and keyboard events as these receive different parameters.

timeout Events registered using the `setTimeout` function. These are events that trigger at specific intervals.

ajax When an asynchronous request to server is send, the programmer can register a handler for an event that is triggered when the server answers.

other The DOM standard defines many specialized events, some rarely used. In addition there exists vendor specific events that are not standardized. This category covers all of those.

There are several things about event handlers that we do not track, most noticeably the node in the DOM tree that the event handler is registered on[2]. This abstraction means that we do not handle the bubbling mechanism where events triggered on a node, bubbles up to event handlers on parent nodes. In our experiments this has not proven to be a major issue, but conceivably an analysis of a framework that depends heavily in event bubbling might be hindered by this limitation.

Actually triggering the events is modeled by changing the flow graph of DOM applications. The top level is turned into a flow graph in the usual fashion, but after executing the top level code, instead of terminating, the flow graph enters the event loop illustrated in Figure 4.3. First, all the registered **load** event handlers are triggered. After that the actual event loop is entered, where a special flow graph node triggers all event handler in the current state, simulating both users, timeout and AJAX events occurring in a non-deterministic order.

[2]Note that since we do not model individual DOM nodes, we would have to extend the modelling of the DOM as well to be more precise here.

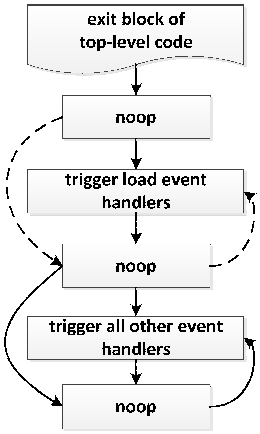


Figure 4.3: Flow graph for the event loop

4.8 The Unevalizer

Richards et al. [78] demonstrated that `eval` is widely used and cannot be ignored when designing JavaScript tools. Dynamic code evaluation presents a big challenge to TAJS as we have soundness as a general design criteria. Since `eval` can execute arbitrary code, it can potentially change any part of the accessible state to any value. Therefore the only sound transfer function we can give for `eval` without modifying TAJS any further is the one that always yields $\top$, which of course is unsatisfactory.

This section will present the approach we take to `eval` in TAJS which is based on refactoring the program during analysis to remove the call to `eval` in a sound manner. The full details are discussed in the paper in Chapter 10.

4.8.1 Measuring `eval` in practice

We know that it is impossible to handle `eval` in the general case without ruining the precision of the analysis. However another conclusion by Richards et al. is that many common uses of `eval` are replaceable by equivalent code that does not use dynamic evaluation. Our goal is to extend TAJS to handle these cases.

Richards et al. shows that a significant portion of `eval` calls are dedicated to either JSON parsing or dynamic code loading. Since most browsers has APIs dedicated to this, `eval` calls used exclusively for JSON parsing can trivially be transformed. Dynamic code loading is outside the scope of TAJS, we assume that all code is available when the analysis is first started.

We divide the remaining cases into `eval` calls which evaluate constant strings (that is strings that appear directly in the source code) and calls that evaluate strings that are constructed dynamically. Using the infrastructure of Richards et al. we examined the Alexa top 10,000 sites. Of those 6465 sites used `eval`, and of those 3378 used `eval` for other than JSON and dynamic code loading. Of those sites, 2589 apply `eval` to truly dynamic strings.

Handling constant strings is comparetively easy compared to the dynamic case. We describe both in the following sections.

4.8.2 Unevalizer Framework

As mentioned the transformation of `eval` calls has to happen during the fixpoint algorithm to ensure soundness. This is done by the Unevalizer component. Whenever TAJS encounters a call to `eval` the Unevalizer component is invoked, if it can successfully refactor the code, the fixpoint iteration continues. If not, it terminates with an error.

The Unevalizer is implemented as a separate component independent of TAJS. It is invoked with all the analysis information it needs to determine if a transformation is possible or not. In the current implementation, the following parameters are passed to the Unevalizer:

- E is the syntactic argument expression as it appears in the program code at the function call site.
- V is the abstract value of the argument expression E. This abstract value soundly approximates the code string to be evaluated.
- D_G and D_L are the sets of variable qnames and function declarations in the global and local scope, respectively. This takes into account nesting of functions and properties of the global object. D_M is the set of names of built-in properties of the global object that may have been modified by the application code. We settle for sound approximations of these sets since JavaScript does not have ordinary lexical scope (due to `with` statements and dynamically constructed properties of the global object that are always in scope).
- r is a boolean flag that indicates whether the call appears syntactically as an expression where its return value is used (as in `x=eval(y)`) or as a statement on its own.
- p is a boolean flag that signals whether the `eval` call is direct or aliased.
- n is a number that indicates the `eval` nesting depth, which is 0 for an `eval` call that occurs in the original source program, 1 for a call that appears in code generated by an `eval` call at nesting depth 0, etc.

Collectively this 8-tuple represents all the information that the Unevalizer needs to do the code transformation. In response the Unevalizer will return either a code string (without calls to `eval` in it) or fail with a special value $\lightning$. Yielding $\lightning$ will cause the analysis to halt as the `eval` call cannot be transformed and thus the program cannot be analyzed.

4.8.3 Constant strings

As mentioned, a significant amount of `eval` calls are dedicated to handling constant strings which we therefore treat separately. Note that "constant" in this regard refers to strings that are either code constants *or* constants at analysis time. For instance the below program illustrates a case where `eval` is not called with an actual code constant but due to TAJS's constant propagation it will be constant at analysis time.

```
1 var json = "<large constant string>";
2 ...
3 eval("area="+json);
```

The value of variable `json` is propagated down to the `eval` call where constant propagation will yield the actual value of the concatenation expression. This is value is then passed to the Unevalizer (the E component above).

At first glance one might think the only thing the Unevalizer would need to do to transform an `eval` call such as `eval("code")` was to return simply `code`. Things are, however, not quite that simple. In the following we will present a few examples of complications that the Unevalizer must handle, the full details are in sections 10.4 and 10.5.

The return value of `eval` is dictated by the code being evaluated. If the code is just a single expression, the return value of `eval` is that expression. In general `eval` can be used to evaluate arbitrary statements (which usually does not yield a value) and this complicates the rules for return value. In JavaScript many statement actually have values and these will be returned from the `eval` call. A few statements, the empty statement and `var` declaration statements for instance, does not yield a value however. The general rule is that the value of an `eval` call is the value of the last value yielding statement executed.

This rule implies that it cannot be statically determined which statement will yield the return value. Take for example the following call to `eval`:

```
1 eval("2;if (b) 3;")
```

Depending on the value of `b` (whose value is read from the surrounding environment) this call will return either 2 or 3.

Note that this presents a problem to transformation only when the return is used in the surrounding context, which except in the case of JSON parsing, is rare. We side step the issue and simply let the Unevalizer return ϟ if the return value is used and the statement that yields the return value of the evaluated code is ambiguous.

4.8.4 Dynamically created strings

In cases where the analysis fails to infer an actual string value for the `eval` argument we are forced to deal with strings that are partially or totally unknown. As we wish to remain sound some information about the arguments must be available, especially if it is a string. If the string is completely unknown, i.e. it is the lattice value String or NotUInt, the transformer component must halt with an error as the concrete value can potentially contain any kind of code.

In many cases, however, the analysis will be able to infer some information about the argument, or parts of the argument if it is a concatenation expression. This extra information will in some cases still allow transformation.

JSON strings

If we know that the string value `s` contains JSON data, then `eval(s)` can be transformed to the equivalent `JSON.parse(s)` which is more safe. To achieve this, the String lattice is extended with a JSONValue indicating data that is known JSON.

JSON data is usually either received via AJAX over the network, or constructed using string operations. Both of these are hard to detect without programmer intervention. Data received over the network are in particular totally opaque to the analysis. We therefor rely on the user to annotate code to

indicating strings that are JSON values and can be transformed in the above described manner.

Common patterns

We found in our study that certain patterns repeat in the way programmers use `eval`. As mentioned, many of these patterns can be transformed to equivalent code without `eval`. To handle this, we extended the transformation component to recognize these patterns and transform them. In addition we identify what are the preconditions required for the transformation to be sound.

Dynamic property lookups are a common superfluous use of `eval`. Take for instance the following code snippet:

```
1 var foo = eval("o." + k)
```

Whether or not this invocation can be transformed depends on the information the analysis has on `k`. If no information is available then the analysis must halt, as `k` could contain arbitrary code. However in practice `k` will most likely be a string denoting a valid property name. If, at runtime, `k` is always a valid property name then the follow transformation is sound:

```
1 var foo = o[k]
```

To be able to verify this precondition, we must therefore extend the analysis so that it can ensure that `k` is in fact a valid identifier. This has been accomplished by extending the string lattice with a new value, IDString, which denotes the set of strings that are also valid identifiers.

A related pattern such as `eval("foo_"+x)` is also quite common in web applications. In this case requiring that `x` is valid identifier is too strong, it only needs to be a valid identifier fragment for this code to be transformable. To handle this case the lattice is extended with the value `IdPartsString` denoting strings that can form parts of valid identifiers.

The pattern `eval("foo_"+x)` is a dynamic lookup in the current scope. JavaScript does not provide a general way to look up computed names in the current scope, unless the current scope is the global one. If the lookup is in the global scope then the following code is a valid transformation:

```
1 (function () {return this})["foo\_"+x]
```

While unwieldy this code is more amenable to further analysis than the alternative version using `eval`

4.9 Related work

This section will review work related to static analysis of JavaScript and how the different works compare with TAJS as described in this chapter. We will also mention some more general results in static analysis in connection with the lazy propagation extension of TAJS. We only discuss works related to static analysis in this section. Section 3.6 discusses approaches to bug finding and program understanding of JavaScript programs that does not involve static analysis.

4.9.1 Static analysis for JavaScript

The analysis builds on large body of work in abstract interpretation and data-flow analysis, and it draws inspiration from dynamic typing. The contribution lies in the combination of known techniques and the implementation supporting the full JavaScript language.

Static analysis of scripting languages has evolved from earlier work on type analysis of dynamically typed languages such as Scheme and Smalltalk. These works have shown the need for a type structure involving union types and recursive types.

Furr et al. [29] have developed a typed dialect of Ruby, a scripting language with features very similar to JavaScript. Their approach requires the programmer to supply type annotations to library functions. Then they employ standard constraint solving techniques to infer types of user-defined functions. There is support for universal types and intersection types (to model overloading), but these types can only be declared, not inferred. They aim for simplicity in favor of precision also to keep the type system manageable, whereas our design aims for precision.

Anderson et al. [3] present a type system with an inference algorithm for a primitive subset of JavaScript based on a notion of definite presence and potential absence of properties in objects. Their system does not model type change and the transition between presence and absence of a property is harder to predict than in a recency-based system.

Zhao [94] builds on the work of Anderson et al. and adds polymorphic types. The system also includes a form of recency typing to allow for strong updates. As with the work of Anderson et al. the type system is defined for small subset of JavaScript which does not include prototypes or dynamic property lookups among others.

The focus in this work is on type inference to bug detection. Other works have focused on detecting security vulnerabilities in AJAX applications [93, 35].

Historically, information gained by static analysis has often been used to optimize code. Not much work has been done on optimizing JavaScript with static analysis yet. Logozzo and Venter present the RATA optimization tool [63] which uses a lightweight static analysis to recognize instances of the JavaScript float type that only holds integers. A compiler or VM can then optimize code with integer specific methods. The types inferred by TAJS could be used to achieve the same result.

Jang and Choe [49] present a points-to analysis for a subset of JavaScript based on set constraints. The results are of the analysis used to optimize property accesses by inlining. Points-to information are part of the abstract state information yielded by TAJS.

Hackett and Guo [38] present a type inference analysis designed to run online in the context of a JIT compiler. The inferred types are used to optimize the code generated by the JIT compiler. The analysis is unsound and uses dynamic checks at runtime to verify the optimizations. Being an online analysis, performance of the algorithm is critical. In contrast, TAJS is designed to be used offline during development, which allows us to put a greater emphasis on precision and soundness.

We have identified JavaScript frameworks such as jQuery as problematic for the analysis. Sridharan et al. [87] identify a code construct termed "correlated dynamic property accesses" that is common within such frameworks. The following code snippet illustrates the pattern:

```
1 function extend(destination, source) {
2   for (var property in source)
3     destination[property] = source[property];
4   return destination ;
5 }
```

The effect of this code is to copy all properties from `destination` to `source`. This presents a problem to most static analyses if the value of `property` can contain multiple possible values this piece of code will conflate different values of the properties on the `source` object. The authors propose a solution called correlation tracking which identifies this pattern using a simple pointer analysis. The code is then transformed by extracting the copy statement into an anonymous function which is then analyzed in a separate context for each property. TAJS has all the necessary infrastructre and could accomdate this technique.

Our transformation of `eval` calls can be viewed as a refactoring that transforms the program to a behaviorally equivalent one without dynamic code evaluation. In a similar way Feldthaus et al. [26] uses a pointer analysis of JavaScript as foundation for a refactoring tool for JavaScript. A crucial difference is that they do transformations after the analysis is completed whereas we are forced to do it during the fixed point computation.

The recency abstraction has turned out to be crucial for the practicality of our analysis. Balakrishnan and Reps [7] were the first to propose the notion of recency in abstract interpretation. They use it to create a sound points-to analysis with sufficient precision to resolve the majority of virtual method calls in compiled C++ code. Heidegger and Thiemann [40] apply this to JavaScript and propose a recency-based type system for a core language of JavaScript.

Our analysis includes context sensitivity, which affects precision. The concrete choice of context was guided by experiments, where we wanted to strike a balance between performance and precision. Other approaches are possible, including variations of the call-string approach.

The IFDS framework by Reps, Horwitz and Sagiv [77] is a powerful approach for obtaining efficient and precise interprocedural analyses. It requires that the lattice underlying the analysis is a power set and the transfer functions be distributive. These requirements are not met by TAJS, and presumably won't be for similar analyses targeting dynamic scripting languages.

Sharir and Pnueli's functional approach to interprocedural analysis can be phrased both with symbolic representations and in an iterative style [85], where the latter is closer to our lazy propagation approach. With the complex lattices and transfer functions that appear to be necessary in analysis for object-oriented scripting languages, symbolic representations are difficult to work with, so TAJS instead uses the iterative style and a relatively direct representation of lattice elements. Furthermore, the functional approach is computationally expensive if the analysis lattice is large.

4.9.2 DOM modeling

As mentioned modeling the DOM has two elements: Supporting the API and modeling the DOM tree in some fashion. The work presented in this dissertation is the first to model the connections between them, previous work has dealt mostly with very coarse models of the API and ignores the HTML page completely.

The Gatekeeper project by Guarnieri and Livshits [33, 32] uses an Anderson-style points-to analysis to analyze the JavaScript code. The results of the analysis is used to verify custom security policies for a web application. The analysis uses a mock-up of the DOM API and ignores the DOM tree.

Guha et al. [35] uses a k-CFA analysis to extract a model of the expected client behavior as seen from the server. This information is then used to generate a checker that can issue warnings when the user behaves in an unexpected manner. 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 data-flow and event handlers in connection to the DOM.

Chugh et al. [16] use staged information flow analysis to protect against dynamic loading of malicious code. 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.

4.9.3 Dynamic code evaluation

Many analysis tools for JavaScript simply ignore the effect of dynamic evaluation. This is acceptable if one focuses on some domain where `eval` is not as prevalent, but if one wishes to analyze web applications, `eval` must be taken into account.

Meawad et al. [66] present the Evalorizer which like the Unevalizer refactors JavaScript code to remove calls to `eval`. Unlike our approach they employ a combined dynamic and static approach. A proxy sits between the browser and the server, monitoring use of `eval`. This collected information is used to recognize common `eval` patterns and transform the code in a similar fashion as our Unevalizer. Given the dynamic approach there is no soundness guarantees. To avoid changing the behavior of the program a recognizer that recognizes the strings for which the transformation is sound is inserted into the code. Invalid strings are passed onto `eval` meaning that calls to `eval` are still present in the code after transformation. The Unevalizer removes calls to `eval` entirely which makes the resulting code more amenable to further analysis.

The `with` statement which dynamically modifies the scope chain in JavaScript can also be problematic to some analyses. The `with` statement is supported by the modeling in TAJS. Park et al. [74] carries out a survey of `with` usage in practice. The survey employs the same infrastructure as the one used in our own `eval` survey, originally created by Richards et al. [78]. The `with` survey shows that a significant portion of websites use `with` although it is not as pervasive as `eval`. The authors describe several common patterns found in the survey and present sound rewriting techniques that removes the `with`

statement. As mentioned, `TAJS` has native support for the `with` statement and while the presented techniques could potentially be adapted to TAJS, the `with` statement has not been shown to be problematic in any of our experiments. Note that many modern browser supports running JavaScript in so-called strict mode which disallows `with` altogether.

In the Gatekeeper project [33] described above the `eval` problem is mitigated by providing a runtime checker that ensure that the JavaScript being executed falls in a safe subset. This subset excludes potentially vulnerable features such as `eval`.

Earlier versions of TAJS used heuristics to handle common `eval` use cases such as simulating higher order functions with strings. This was mostly accomplished by pattern matching and rewriting the code using simple transformation rules. Guha et al. [35] use similar techniques to recognize the dynamic loading of code. Using such heuristics will never be sound and using simple pattern matching on code is inherently fragile.

Furr et al. [29] also treats the problem of dynamic code evaluation in context of the Ruby language. As part of the work they present a transformation from ruby code it an intermediate form amenable to analysis. This transformation is guided by runtime profiling of the code, part of which tries to remove `eval` calls. A key difference with our work is the need for dynamic analysis whereas we rely only on a static approach.

Most dynamic languages such as JavaScript and Ruby support dynamic code evaluation in some form. How problematic `eval` is in a particular language depends on the specific semantics of it. For instance in Scheme [86], `eval` poses less of a problem as code being passed to `eval` is evaluated in an immutable environment, making it safe to ignore or model in a simplistic manner.

Chapter 5

Evaluation

To evaluate the design of TAJS we have carried out a number of experiments that test the various aspects of the tool with regard to both bug detection and program understanding.

As discussed in Section 1.2.2 we must evaluate both precision and performance to get an adequate measure of the usability of the tool. In this section we will outline the research questions and the experimental setup used to investigate each question. For the full details of the experiments we will refer to relevant parts of the papers in Part II.

5.1 Research questions

The following is a complete list of the research question about TAJS that we have experimentally investigated. The list is divided into 4 categories. Each category represent one aspect of the evaluation.

Bug detection

Q1 Can TAJS prove the absence of common JavaScript bugs in already tested JavaScript code? As described in Section 1.2 we do not expect to find many bugs in our benchmarks as they have already been thoroughly tested.

Q2 For programs with errors, can the analysis help the programmer to locate the errors? Specifically, are the warning messages produced by the tool useful toward leading the programmer to the source of the errors? Identifying errors is only useful if the tool can accurately pin-point them and make the programmer aware of it.

Q3 Does the analysis succeed in identifying dead or unreachable code? In some situations, dead or unreachable code is unintended by the programmer and can be considered errors. The ability of the analysis tool to detect such code can in principle also be used to reduce application code size before deployment.

Precision

Q4 How precise is the call graph inferred by the analysis? Having a good approximation of the call graph of a program is a foundation for other potential applications, such as program comprehension or optimization.

Q5 Similarly to the previous question, how precise are the inferred types?

Unevalizer

Q6 Is the Unevalizer able to transform common usage patterns of `eval` calls? This is the main criteria for the usability of the Unevalizer.

Q7 To what extent are the individual techniques presented in sections 4.8.3 and 4.8.4 useful in practice? The extra techniques add complexity so to be worth the extra overhead they must increase the number of handled cases.

Q8 For call sites where the Unevalizer fails to find a valid transformation, can we suggest improvements that are likely to handle more cases?

Lazy propagation

Q9 Does lazy propagation have a noticeable impact on performance of the analysis? Lazy propagation introduces extra overhead which could potentially cancel out any gains from the technique.

Q10 Does lazy propagation improve the precision of the analysis? The possible precision improvements from lazy propagation are discussed in Section 4.6.2.

5.2 Results

This sections summarizes the experiments carried out to answer the above questions and the results we obtained.

Bug detection

Q1 is addressed in two separate experiments detailed in sections 7.5 and 9.5, one addressing the modeling of the core JavaScript semantics and one addressing the DOM modeling. We test the core modeling on a collection of standalone benchmarks that do not use DOM or any other API not present in the ECMAScript specification. The benchmarks are drawn from the Google V8[1] and SunSpider[2] benchmark suites. As mentioned we measure the percentage of program locations where TAJS is able to prove the absence of specific errors. The following list describes the errors that the analysis looks for in this experiment:

- Invoking a non-function value as either a regular function or a constructor.
- Reading an uninitialized variable.
- Accessing an undefined property of an object using a dynamically computed property name.

[1] `http://v8.googlecode.com/svn/data/benchmarks/v3/run.html`
[2] `http://www.webkit.org/perf/sunspider/sunspider.html`

- Accessing an undefined property using a static property name.

The last two entries are not actual errors that will cause the program to crash. They are, however, situations that indicate a possible programmer error and therefore the tool should make the programmer aware of them. To measure the effectiveness of TAJS on applications using the DOM we use a collection of web applications that use the DOM drawn from Chrome Experiments[3], IE Test Drive[4] and 10k apart[5]. We run TAJS on these programs and try to verify the absence of the errors outlined above.

The results are encouraging. In most cases the analysis is able to prove between 80-100% of the relevant program points as error free. The results are discussed in detail in sections 7.5 and 9.5.3.

Note that the experiments done in connection with the paper in Chapter 7 were carried out with a version of TAJS that did not have the lazy propagation extension.

To answer **Q2** we introduce errors into the benchmark programs at random. To simulate spelling errors made by the programmer we pick a random read or write property operation that uses the fixed-property notation (i.e. the . operator) and replace 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.

Research question **Q3** asks if the analysis can detect dead and unreachable code. Dead code is code without any effect, such as an assignment to variable that is never subsequently read. Unreachable code is parts of the program that will never get executed. While neither are erroneous, both can indicate a bug. To measure unreachable code we simply count the number of functions that TAJS determines to be unreachable. Likewise, to measure dead code we count the number of operations that the analysis can prove to be dead. In the results we see that TAJS is able to identify both dead and unreachable code in many benchmarks. Most of the unreachable or dead code being detected appears to be code left from earlier revisions of the programs. Refer to Section 9.5 for detailed results.

[3] `http://www.chromeexperiments.com/`
[4] `http://ie.microsoft.com/testdrive/`
[5] `http://10k.aneventapart.com/`

Precision

Regarding **Q4**: Measuring the precision of the resulting call graph is one aspect of measuring the overall precision of the analysis. We measure this by calculating the ratio of monomorphic call sites compared to the total number of call sites. A call site is monomorphic if it has only one invocation target. The results are surprising: For 49 out of 53 benchmarks the number is 100% which shows that even though JavaScript supports higher order functions, most function calls are monomorphic. This also demonstrates that TAJS is precise enough to actually show this fact.

For **Q5** we wish to measure the precision of the computed types. TAJS tracks the following kinds of abstract values: boolean, number, string, object (including null and function values) and the special type undefined (for more details on types see Section 3.1). An object property can therefore potentially hold values of up to five different types. We measure the accuracy of the inferred types by calculating the average number of different types present in the abstract values resulting from read operations. We take the average of all read operations in the benchmark except ones deemed unreachable by the analysis. If this number is 1 then every read operation results in abstract values of a unique type in all possible executions. In the results we see a similar pattern as with **Q4**. Even though JavaScript supports dynamic variables TAJS is able to show that the majority of read operations only yield values of one type. Of the 26,870 property read operations that appear in the benchmarks, the analysis finds that at most 4,019 can have multiple types.

Unevalizer

The next research questions deal with the Unevalizer component and its integration with TAJS. To evaluate this aspect we must have benchmarks that use dynamic evaluation such as the `eval` function. As described in Section 4.8.1 we studied a large set of web pages using JavaScript to determine how they use `eval`. We also draw our benchmarks from this set. We focus on the most challenging cases of `eval`, which are the call sites that fall into the categories "other" or "single operation" described in Section 4.8.1. We exclude all web sites that do not have any instances of `eval` in these categories. Applying these criteria on the Alexa top 500 list gives us 19 web sites.

These programs are full sized web applications and while TAJS can handle many real world programs, these are beyond its current capabilities. To test the performance of the Unevalizer, we therefore manually extract the interesting parts of the identified programs. This process yields 28 "program slices" where each slice is a small self-contained program that use dynamic code evaluation. See Section 10.6.1 for a further discussion of the benchmarks and how they were selected.

To answer **Q6** we run TAJS with the Unevalizer on all the 28 benchmarks. If TAJS is able to reach a fixpoint on a given benchmark it counts as a success otherwise it is a failure. TAJS fails to reach a fixpoint if the Unevalizer fails to transform a given `eval` call. We see that the Unevalizer is able to handle 19 out of 28 cases, corresponding to 33 out of 44 `eval` call sites.

We address **Q7** by counting how often the different techniques presented in Sections 4.8.3 and 4.8.4 are needed to soundly transform a call site. We see

that out of 44 call sites, constant propagation alone is enough to transform 15 eval call sites. Using identifier detection we eliminate 9 more call sites and if we also add specialization, 9 additional call sites are successfully transformed. These numbers suggest that all the techniques we have presented are useful in practice. To see which techniques worked for each individual benchmark refer to Section 10.6.2.

Finally **Q8** asks what, if anything, can be done in the cases where the Unevalizer fails to transform a call site. We have studied these cases and we see that the main reason for failure is loops where the analysis loses too much precision in the loop body.

The following code snippet illustrates the problem:

```
1 for (var libName in iTXT.js.loader) {
2   currentLibName = libName;
3   eval(libName + '_Load()');
4 }
```

The loop iterates over all the properties of an object which is defined as a constant object literal elsewhere in the code. These property names do not match the special lattice value IDString (see Section 4.8.4) so the abstract value of `libName` becomes $\top$ which is insufficient to check the preconditions required by the Unevalizer. Applying loop unrolling would enable the analysis to do better constant propagation, which would in turn enable the Unevalizer to transform the call site.

Lazy propagation

Properties of lazy propagation are investigated when answering **Q9** and **Q10**. We run TAJS on the set of standalone benchmarks used to answer **Q1**. We do this both with and without the lazy propagation technique enabled and compare the results. The most robust measurement of analysis performance is the number of iterations used to reach a fixpoint as this is independent of the environment the analysis is run in. In addition to iterations, we also measure execution time and memory consumption. The results uniformly indicate that lazy propagation is worth the extra overhead. The number of iterations are reduced significantly, at times by over 50%. Both execution time and memory consumption is also reduced by enabling lazy propagation.

Q10 deals with the precision of using lazy propagation. On the set of benchmarks collected we see no noticeable improvement of precision when using lazy propagation.

See Section 8.4 for more details on the experimental evaluation of lazy propagation.

A limitation that is revealed by our experiments is the inability of TAJS to handle JavaScript frameworks. These are often highly complex and make extensive use of the more dynamic features of JavaScript. When analyzing applications that use these libraries TAJS loses too much precision and the analysis fails to reach a fixpoint before running of out memory.

5.3 Threats to validity

The programs we have picked for our experiments are written by many different programmers and exhibit different characteristics. They exhibit a large variety of the functionality supported by JavaScript and the DOM API and our experiments show that the analysis is able to infer detailed and useful type information about them. The main limiting factor in choosing these benchmarks has been program size and the presence of JavaScript frameworks (see Section 2.4 for at description of JavaScript frameworks). The focus of this work has been on creating a precise and sound modeling of JavaScript and connected technologies with performance as a second priority. Our implementation, TAJS, is in its current state not able to handle very large applications or applications that make heavy use of libraries such as jQuery or prototype.

The benchmarks have been selected with these restrictions in mind. We have chosen small to medium sized standalone benchmarks that do not rely on libraries. In the case of the Unevalizer we have applied manual slicing to extract benchmarks that fulfill the same criteria from larger programs.

The central validity threat is therefore that the benchmarks we have chosen for evaluation are not representative of the JavaScript programs that are produced by the majority of programmers because they are either too small or not complex enough.

While the current size limitation of TAJS is a problem for the applicability of the tool, scalability has not been the primary focus of the research carried out in this dissertation. We see improving performance is a definite avenue for future work.

Chapter 6

Conclusion

We have presented a data-flow analysis for the JavaScript language. The analysis can determine detailed type information about JavaScript programs including points-to information and call graphs. The main contribution of the core analysis is the design of the lattice of abstract values. The design of the lattice is the result of an experimentally driven process and is an attempt to strike a balance between precision and performance. The analysis is a sound modeling of the JavaScript semantics as laid out in the specification [23].

To improve the performance of the analysis we have developed the lazy propagation technique. Lazy propagation is based on the observation that the analysis spends a significant amount of time propagating abstract states in the fixpoint computation. Often times this work is redundant. Lazy propagation changes the analysis to only propagate parts of the state that is known to be used. This reduces both the number of iterations needed to reach a fixpoint and the size of the abstract states, leading to improved performance of the analysis.

To handle JavaScript programs running on the web, the analysis is instrumented to model both the browser and the DOM API. This is implemented by a lattice extension to track event handlers and a set of transfer functions to model the API. Furthermore an abstract model of the HTML page containing the JavaScript is added to the initial environment.

Dynamic code evaluation presents a significant challenge to static analysis design. Surveys of real code shows that many uses of dynamic code evaluation follow the same patterns. Many of these patterns can be transformed into equivalent code that does not use dynamic evaluation and is therefor easier to analyze. We have designed the Unevalizer component which can recognize and transform these code patterns. To remain sound, the transformation is done during the fixpoint computation and the Unevalizer leverages analysis information to ensure that the code resulting from the transformation is in fact equivalent.

We have experimentally validated the hypothesis that we described in Section 1.1 using our implementation of the analysis, TAJS. To validate the hypothesis we must show that TAJS has both sufficient precision and performance to detect bugs in programs.

Our experiments show that TAJS yields results with sufficient precision to prove the absence of common errors in JavaScript code. The programs tested

include DOM applications and pure JavaScript applications that do not rely on any framework libraries. On a majority of these applications TAJS achieves 80% accuracy which we consider satisfactory.

In addition we have also evaluated the precision of the abstract values and the call graph inferred by TAJS. Both the resulting abstract values and call graph have high precision and would presumably be useful for program comprehension in an IDE for instance.

Performance-wise our experiments show that the lazy propagation technique provides a significant improvement on both execution time and memory consumption of the analysis. However, TAJS is still limited to medium sized JavaScript applications. JavaScript frameworks such as JQuery are still not possible to analyze with TAJS. Given how widespread such frameworks are this a major hindrance in the application of TAJS to real programs. Techniques such as the one proposed by Sridharan et al. [87] could potentially be adapted to TAJS to improve upon this situation.

To summarize, the following scientific contributions has been made as a result of the work presented in this dissertation:

- Even though JavaScript is a complicated dynamic language we have showed that it is possible to do static analysis which is both precise and reasonably performant.
- During the design of the analysis we developed the lazy propagation optimization technique. Lazy propagation reduces the size of abstract states which is especially beneficial to our analysis which uses complex lattices.
- We describe how to model the Document Object Model and browser event loop. This allows us to reason about data and control flow in JavaScript web applications.
- We show how to automatically eliminate calls to `eval` soundly by refactoring the code during the fixpoint computation. Furthermore we identify common patterns of `eval` usage and the preconditions needed to soundly refactor them.

We conclude that for a large class of applications, TAJS provides high enough precision to detect common bugs in the code. For this class of applications performance is acceptable. However, for large applications the current design proves inadequate. Further work is needed to investigate the reasons and possible solutions for the problems encountered.

Part II

Papers

Chapter 7

Type Analysis for JavaScript

Abstract

JavaScript is the main scripting language for Web browsers, and it is essential to modern Web applications. Programmers have started using it for writing complex applications, but there is still little tool support available during development.

We present a static program analysis infrastructure that can infer detailed and sound type information for JavaScript programs using abstract interpretation. The analysis is designed to support the full language as defined in the ECMAScript standard, including its peculiar object model and all built-in functions. The analysis results can be used to detect common programming errors – or rather, prove their absence, and for producing type information for program comprehension.

Preliminary experiments conducted on real-life JavaScript code indicate that the approach is promising regarding analysis precision on small and medium size programs, which constitute the majority of JavaScript applications. With potential for further improvement, we propose the analysis as a foundation for building tools that can aid JavaScript programmers.

7.1 Introduction

In 1995, Netscape announced JavaScript as an "easy-to-use object scripting language designed for creating live online applications that link together objects and resources on both clients and servers" [67]. Since then, it has become the de facto standard for client-side scripting in Web browsers but many other applications also include a JavaScript engine. This prevalence has lead developers to write large programs in a language which has been conceived for scripting, but not for programming in the large. Hence, tool support is badly needed to help debug and maintain these programs.

The development of sound programming tools that go beyond checking mere syntactic properties requires some sort of program analysis. In particular, type analysis is crucial to catch representation errors, which e.g. confuse numbers with strings or booleans with functions, early in the development process. Type analysis is a valuable tool to a programmer because it rules out this class of programming errors entirely.

Applying type analysis to JavaScript is a subtle business because, like most other scripting languages, JavaScript has a weak, dynamic typing discipline which resolves many representation mismatches by silent type conversions. As JavaScript supports objects, first-class functions, and exceptions, tracking the flow of data and control is nontrivial. Moreover, JavaScript's peculiarities present a number of challenges that set it apart from most other programming languages:

- JavaScript is an object-based language that uses prototype objects to model inheritance. As virtually all predefined operations are accessed via prototype objects, it is imperative that the analysis models these objects precisely.
- Objects are mappings from strings (property names) to values. In general, properties can be added and removed during execution and property names may be dynamically computed.
- Undefined results, such as accessing a non-existing property of an object, are represented by a particular value `undefined`, but there is a subtle distinction between an object that lacks a property and an object that has the property set to `undefined`.
- Values are freely converted from one type to another type with few exceptions. In fact, there are only a few cases where no automatic conversion applies: the values `null` and `undefined` cannot be converted to objects and only function values can be invoked as functions. Some of the automatic conversions are non-intuitive and programmers should be aware of them.
- The language distinguishes primitive values and wrapped primitive values, which behave subtly different in certain circumstances.
- Variables can be created by simple assignments without explicit declarations, but an attempt to read an absent variable results in a runtime error. JavaScript's `with` statement breaks ordinary lexical scoping rules, so even resolving variable names is a nontrivial task.
- Object properties can have attributes, like ReadOnly. These attributes cannot be changed by programs but they must be taken into account by the analysis to maintain soundness and precision.
- Functions can be created and called with variable numbers of parameters.
- Function objects serve as first-class functions, methods, and constructors with subtly different behavior. An analysis must keep these uses apart and detect initialization patterns.
- With the `eval` function, a dynamically constructed string can be interpreted as a program fragment and executed in the current scope.
- The language includes features that prescribe certain structures (the global object, activation objects, argument objects) in the implementation of the runtime system. These structures must be modeled in an analysis to obtain sufficient precision.

This paper reports on the design and implementation of a program analyzer for the full JavaScript language. In principle, the design is an application of abstract interpretation using the monotone framework [17, 56]. However,

the challenges explained above result in a complicated lattice structure that forms the basis of our analysis. Starting from a simple type lattice, the lattice has evolved in a number of steps driven by an observed lack of precision on small test cases. As the lattice includes precise singleton values, the analyzer duplicates a large amount of the functionality of a JavaScript interpreter including the implementation of predefined functions. Operating efficiently on the elements of the lattice is another non-trivial challenge.

The analyzer is targeted at hand-written programs consisting of a few thousand lines of code. We conjecture that most existing JavaScript programs fit into this category.

One key requirement of the analysis is *soundness*. Although several recent bug finding tools for other languages sacrifice soundness to obtain fewer false positives [11, 24], soundness enables our analysis to guarantee the absence of certain errors. Moreover, the analysis is *fully automatic*. It neither requires program annotations nor formal specifications.

While some programming errors result in exceptions being thrown, other errors are masked by dynamic type conversion and `undefined` values. Some of these conversions appear unintuitive in isolation but make sense in certain circumstances and some programmers may deliberately exploit such behavior, so there is no clear-cut definition of what constitutes an "error". Nevertheless, we choose to draw the programmer's attention to such potential errors. These situations include

1. invoking a non-function value (e.g. `undefined`) as a function,
2. reading an absent variable,
3. accessing a property of `null` or `undefined`,
4. reading an absent property of an object,
5. writing to variables or object properties that are never read,
6. implicitly converting a primitive value to an object (as an example, the primitive value `false` may be converted into a `Boolean` object, and later converting that back to a primitive value results in `true`, which surprises many JavaScript programmers),
7. implicitly converting `undefined` to a number (which yields `NaN` that often triggers undesired behavior in arithmetic operations),
8. calling a function object both as a function and as a constructor (i.e. perhaps forgetting `new`) or passing function parameters with varying types (e.g. at one place passing a number and another place passing a string or no value),
9. calling a built-in function with an invalid number of parameters (which may result in runtime errors, unlike the situation for user defined functions) or with a parameter of an unexpected type (e.g. the second parameter to the `apply` function must be an array).

The first three on this list cause runtime errors (exceptions) if the operation in concern is ever executed, so these warnings have a higher priority than the others. In many situations, the analysis can report a warning as a definite error rather than a potential error. For example, the analysis may detect that a property read operation will always result in `undefined` because the given property is never present, in which case that specific warning gets high priority.

As the analysis is sound, the absence of errors and warnings guarantees that the operations concerned will not fail. The analysis can also detect dead code.

The following tiny but convoluted program shows one way of using JavaScript's prototype mechanism to model inheritance:

```
function Person(n) {
  this.setName(n);
  Person.prototype.count++;
}
Person.prototype.count = 0;
Person.prototype.setName = function(n) { this.name = n; }
function Student(n,s) {
  this.b = Person;
  this.b(n);
  delete this.b;
  this.studentid = s.toString();
}
Student.prototype = new Person;
```

The code defines two "classes" with constructors `Person` and `Student`. `Person` has a static field `count` and a method `setName`. `Student` inherits `count` and `setName` and defines an additional `studentid` field. The definition and deletion of `b` in `Student` invokes the super class constructor `Person`. A small test case illustrates its behavior:

```
var t = 100026.0;
var x = new Student("Joe Average", t++);
var y = new Student("John Doe", t);
y.setName("John Q. Doe");
assert(x.name === "Joe Average");
assert(y.name === "John Q. Doe");
assert(y.studentid === "100027");
assert(x.count == 3);
```

Even for a tiny program like this, many things could go wrong – keeping the different errors discussed above in mind – but our analysis is able to prove that none of the errors can occur here. Due to the forgiving nature of JavaScript, errors may surface only as mysterious `undefined` values. Simple errors, like misspelling `prototype` or `name` in just a single place or writing `toString` instead of `toString()`, are detected by the static type analysis instead of causing failure at runtime. The warning messages being produced by the analysis can help the programmer not only to detect errors early but also to pinpoint their cause.

Contributions

This work is the first step towards a full-blown JavaScript program analyzer, which can be incorporated into an IDE to supply on-the-fly error detection as well as support for auto-completion and documentation hints. It focuses on JavaScript version 1.5, corresponding to ECMAScript 3rd edition [23], which is currently the most widely used variant of the language and which is a subset of the upcoming revision of the JavaScript language.

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

- We define a type analysis for JavaScript based on abstract interpretation [17]. Its main contribution is the design of an intricate lattice structure that fits with the peculiarities of the language. We design the analysis building on existing techniques, in particular recency abstraction [7].
- We describe our prototype implementation of the analysis, which covers the full JavaScript language as specified in the ECMAScript standard [23], and we report on preliminary experiments on real-life benchmark programs and measure the effectiveness of the various analysis techniques being used.
- We identify opportunities for further improvements of precision and speed of the analysis, and we discuss the potential for additional applications of the analysis technique.

Additional information about the project is available online at

`http://www.brics.dk/TAJS`

7.2 Related Work

The present work builds on a large body of work and experience in abstract interpretation and draws inspiration from work on soft typing and dynamic typing. The main novelty consists of the way it combines known techniques, leading to the construction of the first full-scale implementation of a high precision program analyzer for JavaScript. It thus forms the basis to further investigate the applicability of techniques in this new domain.

Dolby [22] explains the need for program analysis for scripting languages to support the interactive completion and error spotting facilities of an IDE. He sketches the design of the WALA framework [27], which is an adaptable program analysis framework suitable for a range of languages, including Java, JavaScript, Python, and PHP. While our first prototype was built on parts of the WALA framework, we found that the idiosyncrasies of the JavaScript language required more radical changes than were anticipated in WALA's design.

Eclipse includes JSDT [14], which mainly focuses on providing instantaneous documentation and provides many shortcuts for common programming and documentation patterns as well as some refactoring operations. It also features some unspecified kind of prototype-aware flow analysis to predict object types and thus enable primitive completion of property names. JSEclipse [1] is another Eclipse plugin, which includes built-in knowledge about some popular JavaScript frameworks and uses the Rhino JavaScript engine to run parts of the code to improve support for code completion. Neither of these plugins can generate warnings for unintended conversions or other errors discussed above.

Program analysis for scripting languages has evolved from earlier work on type analysis for dynamically typed languages like Scheme and Smalltalk [12, 92, 31]. These works have clarified the need for a type structure involving union types and recursive types. They issue warnings and insert dynamic tests in programs that cannot be type checked. MrSpidey [28] is a flow-based implementation of these ideas with visual feedback about the location of the checks in a programming environment. In contrast, our analysis only reports warnings because the usefulness of checks is not clear in a weakly typed setting.

Thiemann's typing framework for JavaScript programs [89] has inspired the design of the abstract domain for the present work. That work concentrates

on the design and soundness proof, but does not present a typing algorithm. In later work, Heidegger and Thiemann [40] propose a recency-based type system for a core language of JavaScript, present its soundness proof, sketch an inference algorithm, and argue the usefulness of this concept.

Anderson and others [3] present a type system with an inference algorithm for a primitive subset of JavaScript based on a notion of definite presence and potential absence of properties in objects. Their system does not model type change and the transition between presence and absence of a property is harder to predict than in a recency-based system.

Furr and others [29] have developed a typed dialect of Ruby, a scripting language with features very similar to JavaScript. Their approach requires the programmer to supply type annotations to library functions. Then they employ standard constraint solving techniques to infer types of user-defined functions. There is support for universal types and intersection types (to model overloading), but these types can only be declared, not inferred. They aim for simplicity in favor of precision also to keep the type language manageable, whereas our design aims for precision. Their paper contains a good overview of further, more pragmatic approaches to typing for scripting languages like Ruby and Python.

Similar techniques have been applied to the Erlang language by Marlow and Wadler [65] as well as by Nyström [71]. These ideas have been extended and implemented in a practical tool by Lindahl and Sagonas [61]. Their work builds on success typings, a notion which seems closely related to abstract interpretation.

One program analysis that has been developed particularly for JavaScript is points-to analysis [49]. The goal of that analysis is not program understanding, but enabling program optimization. The paper demonstrates that the results from the analysis enable partial redundancy elimination. The analysis is flow and context insensitive and it is limited to a small first-order core language. In contrast, our analysis framework deals with the entire language and performs points-to analysis as part of the type analysis. As our analysis is flow and context sensitive, it yields more precise results than the dedicated points-to analysis.

Balakrishnan and Reps [7] were first to propose the notion of recency in abstract interpretation. They use it to create a sound points-to analysis with sufficient precision to resolve the majority of virtual method calls in compiled C++ code. Like ourselves, they note that context sensitivity is indispensable in the presence of recency abstraction. However, the rest of their framework is substantially different as it is targeted to analyzing binary code. Its value representation is based on a stride domain and the interprocedural part uses a standard k-limited call-chain abstraction.

Shape analysis [82] is yet more powerful than recency abstraction. For example, it can recover strongly updatable abstractions for list elements from a summary description of a list data structure. This capability is beyond recency abstraction. However, the superior precision of shape analysis requires a much more resource-intensive implementation.

Finally, our analysis uses abstract garbage collection. This notion has been investigated in depth in a polyvariant setting by Might and Shivers [68], who attribute its origin to Jagannathan and others [48]. They, as well as Balakr-

ishnan and Reps [7], also propose abstract counting which is not integrated in our work as the pay-off is not yet clear.

7.3 Flow Graphs for JavaScript

The analysis represents a JavaScript program as a flow graph, in which each node contains an instruction and each edge represents potential control flow between instructions in the program. The graph has a designated program entry node corresponding to the first instruction of the global code in the program. Instructions refer to *temporary variables*, which have no counterpart in JavaScript, but which are introduced by the analyzer when breaking down composite expressions and statements to instructions. The nodes can have different kinds:

declare-variable$[x]$: declares a program variable named x with value `undefined`.

read-variable$[x, v]$: reads the value of a program variable named x into a temporary variable v.

write-variable$[v, x]$: writes the value of a temporary variable v into a program variable named x.

constant$[c, v]$: assigns a constant value c to the temporary variable v.

read-property$[v_1, v_2, v_3]$: performs an object property lookup, where v_1 holds the base object, v_2 holds the property name, and v_3 gets the resulting value.

write-property$[v_1, v_2, v_3]$: performs an object property write, where v_1 holds the base object, v_2 holds the property name, and v_3 holds the value to be written.

delete-property$[v_1, v_2, v_3]$: deletes an object property, where v_1 holds the base object, v_2 holds the property name, and v_3 gets the resulting value.

if$[v]$: represents conditional flow for e.g. `if` and `while` statements.

entry$[f, x_1, \ldots, x_n]$, exit, and exit-exc: used for marking the unique entry and exit (normal/exceptional) of a function body. Here, f is the (optional) function name, and $x_1, \ldots, x_n$ are formal parameters.

call$[w, v_0, \ldots, v_n]$, construct$[w, v_0, \ldots, v_n]$, and after-call$[v]$: A function call is represented by a pair of a call node and an after-call node. For a call node, w holds the function value and $v_0, \ldots, v_n$ hold the values of `this` and the parameters. An after-call node is returned to after the call and contains a single variable for the returned value. The construct nodes are similar to call nodes and are used for `new` expressions.

return$[v]$: a function return.

throw$[v]$ and catch$[x]$: represent `throw` statements and entries of `catch` blocks.

<*op*>$[v_1, v_2]$ and <*op*>$[v_1, v_2, v_3]$: represent unary and binary operators, where the result is stored in v_2 or v_3, respectively.

This instruction set is reminiscent of the bytecode language used in some interpreters [47] but tailored to program analysis. Due to the limited space, we here omit the instructions related to `for-in` and `with` blocks and settle for this informal description of the central instructions. They closely correspond to the

ECMAScript specification – for example, read-property is essentially the [[Get]] operation from the specification.

We distinguish between different kinds of edges. *Ordinary* edges correspond to intra-procedural control flow. These edges may be labeled to distinguish branches at if nodes. Each node that may raise an exception has an *exception* edge to a catch node or an exit-exc node. Finally, *call* and *return* edges describe flow from call or construct nodes to entry nodes and from exit nodes to after-call nodes.

All nodes as well as ordinary edges and exception edges are created before the fixpoint iteration starts, whereas the call and return edges are added on the fly when data flow is discovered, as explained in Section 4.3.

7.4 The Analysis Lattice and Transfer Functions

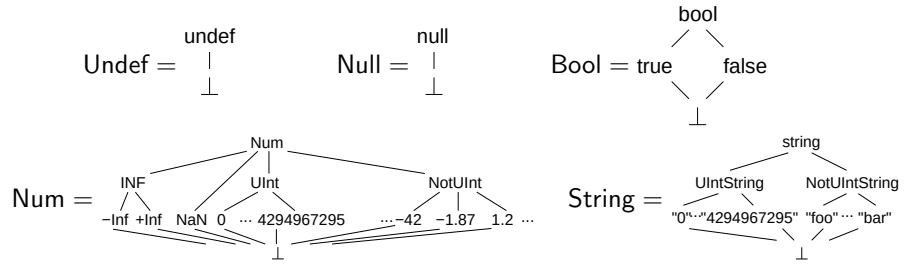
The classical approach of abstract interpretation [17] and the monotone framework [56] requires a lattice of abstract states. Our lattice structure is similar to a lattice used for constant propagation with JavaScript's type structure on top. Numbers and strings are further refined to recognize array indices. For objects, the analysis performs a context-sensitive flow analysis that discovers points-to information.

For a given flow graph, we let N denote the set of nodes, T is the set of temporary variables, and L is the set of *object labels* corresponding to the possible allocation sites (including construct nodes, constant nodes for function declarations, and objects defined in the standard library).

Abstract values are described by the lattice Value:

$$\mathsf{Value} \quad = \quad \mathsf{Undef} \times \mathsf{Null} \times \mathsf{Bool} \times \mathsf{Num} \times \mathsf{String} \times \mathcal{P}(L)$$

The components of Value describe the different types of values.



For example, the abstract value $(\bot, \mathsf{null}, \bot, \bot, \mathtt{baz}, \emptyset)$ describes a concrete value that is either `null` or the string "`baz`", and $(\mathsf{undef}, \bot, \bot, \bot, \bot, \{\ell_{42}, \ell_{87}\})$ describes a value that is `undefined` or an object originating from ℓ_{42} or ℓ_{87}.

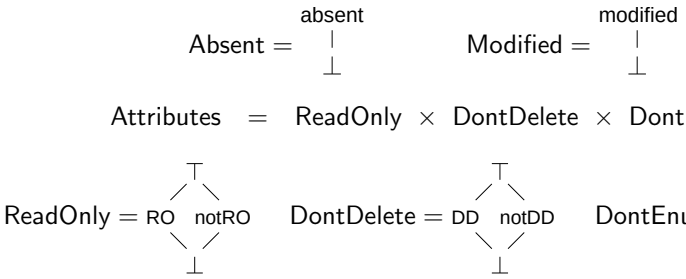
Objects are modeled as follows:

$$\mathsf{Obj} \quad = \quad (P \hookrightarrow \mathsf{Value} \times \mathsf{Absent} \times \mathsf{Attributes} \times \mathsf{Modified}) \times \mathcal{P}(\mathsf{ScopeChain})$$

Here, P is the infinite set of property names (i.e. all strings). The partial map provides an abstract value for every possible property name. There are four special property names: [[Prototype]], [[Value]], *default_index*, and *default_other*. The former two correspond to the internal properties used by ECMAScript; *default_index* and *default_other* are always in the domain of the map and provide an abstract value for all property names that are not in the domain of

the map (hence the map is effectively total): *default_index* covers property names that match UIntString (array indices), and *default_other* covers all other strings. This distinction is crucial when analyzing programs involving array operations. Section 7.4.3 explains the ScopeChain component, which models the special internal property [[Scope]].

Each value stored in an object has additional components. Absent models potentially absent properties, Modified is related to interprocedural analysis as explained in Section 7.4.3, and Attributes models the property attributes ReadOnly, DontDelete, and DontEnum.



An abstract state consists of an abstract store, which is a partial map from object labels to abstract objects, together with an abstract stack:

$$\mathsf{State} \;=\; (L \hookrightarrow \mathsf{Obj}) \;\times\; \mathsf{Stack} \;\times\; \mathcal{P}(L) \;\times\; \mathcal{P}(L)$$

The last two object label sets in State are explained in Section 7.4.3.

The stack is modeled as follows:

$$\begin{aligned} \mathsf{Stack} &= (T \to \mathsf{Value}) \times \mathcal{P}(\mathsf{ExecutionContext}) \times \mathcal{P}(L) \\ \mathsf{ExecutionContext} &= \mathsf{ScopeChain} \times L \times L \\ \mathsf{ScopeChain} &= L^* \end{aligned}$$

The first component of Stack provides values for the temporary variables. The $\mathcal{P}$(ExecutionContext) component models the top-most execution context[1] and the $\mathcal{P}(L)$ component contains object labels of all references in the stack. An execution context contains a scope chain, which is here a sequence of object labels, together with two additional object labels that identify the variable object and the `this` object.

Finally, we define the analysis lattice, which assigns a set of abstract states to each node (corresponding to the program points *before* the nodes):

$$\mathsf{AnalysisLattice} = V \times N \;\to\; \mathsf{State}$$

V is the set of version names of abstract states for implementing context sensitivity. As a simple heuristic, we currently keep two abstract states separate if they have different values for `this`, which we model by $V = \mathcal{P}(L)$.

The lattice order is defined as follows: For the components of Value, the Hasse diagrams define the lattice order for each component. All maps and products are ordered pointwise, and power sets are ordered by subset inclusion

[1] The ECMAScript standard [23] calls a stack frame an *execution context* and also defines the terms *scope chain* and *variable object*.

– except the last $\mathcal{P}(L)$ component of State, which uses $\supseteq$ instead of $\subseteq$ (see Section 7.4.3).

These definitions are the culmination of tedious twiddling and experimentation. Note, for example, that for two abstract stores σ_1 and σ_2 where $\sigma_1(\ell)$ is undefined and $\sigma_2(\ell)$ is defined (i.e. the object ℓ is absent in the former and present in the latter), the join simply takes the content of ℓ from σ_2, i.e. $(\sigma_1 \sqcup \sigma_2)(\ell) = \sigma_2(\ell)$, as desired. Also, for every abstract store σ and every ℓ where $\sigma(\ell) = (\omega, s)$ is defined, we have absent set in $\omega(\textit{default\_index})$ and in $\omega(\textit{default\_other})$ to reflect the fact that in every object, *some* properties are absent. Thereby, joining two stores where an object ℓ is present in both but some property p is only present in one (and mapped to the bottom Value in the other) results in a store where ℓ is present and p is marked as absent (meaning that it is *maybe* absent).

The analysis proceeds by fixpoint iteration, as in the classical monotone framework, using the transfer functions described in Section 7.4.1. The initial abstract state for the program entry node consists of 161 abstract objects (mostly function objects) defined in the standard library.

We omit a formal description of the abstraction/concretization relation between the ECMAScript specification and this abstract interpretation lattice. However, we note that during fixpoint iteration, an abstract state never has dangling references (i.e. in every abstract state σ, every object label ℓ that appears anywhere within σ is always in the domain of the store component of σ). With this invariant in place, it should be clear how every abstract state describes a set of concrete states.

The detailed models of object structures represented in an abstract state allows us to perform *abstract garbage collection* [68]. An object ℓ can safely be removed from the store unless ℓ is reachable from the abstract call stack. This technique may improve both performance and precision (see Section 7.5).

Section 7.5 contains an illustration of the single abstract state appearing at the final node of the example program after the fixpoint is reached.

7.4.1 Transfer Functions

For each kind of node n in the flow graph, a monotone transfer function maps an abstract state before n to a abstract state after n. In addition, we provide a transfer function for each predefined function in the ECMAScript standard library. Some edges (in particular, call and return edges) also carry transfer functions. As usual, the before state of node n is the join of the after states of all predecessors of n.

The transfer function for read-property$[v_{obj}, v_{prop}, v_{target}]$ serves as an illustrative example. If v_{obj} is not an object, it gets converted into one. If v_{obj} abstracts many objects, then the result is the join of reading all of them. The read operation for a single abstract object descends the prototype chain and joins the results of looking up the property until the property was definitely present in a prototype. If v_{prop} is not a specific string, then the *default_index* and *default_other* fields of the object and its prototypes are also considered. Finally, the temporary variable v_{target} is overwritten with the result; all temporaries can be strongly updated. As this example indicates, it is essential that the analysis models all aspects of the JavaScript execution model, including prototype chains and type coercions.

A special case is the transfer function for the built-in functions `eval` and `Function` that dynamically construct new program code. The analyzer cannot model such a dynamic extension of the program because the fixpoint solver requires N and L to be fixed. Hence, the analyzer issues a warning if these functions are used. This approach is likely satisfactory as these functions are mostly used in stylized ways, e.g. for JSON data, according to a study of existing JavaScript code [59].

7.4.2 Recency Abstraction

A common pattern in JavaScript code is creating an object with a constructor function that adds properties to the object using write-property operations. In general, an abstract object may describe multiple concrete objects, so such operations must be modeled with weak updates of the relevant abstract objects. Subsequent read-property operations then read potentially absent properties, which quickly leads to a proliferation of `undefined` values, resulting in poor analysis precision. Fortunately, a solution exists which fits perfectly with our analysis framework: *recency abstraction* [7].

In essence, each allocation site ℓ (in particular, those identified by the construct instructions) is described by *two* object labels: $\ell^{@}$ (called the *singleton*) always describes exactly one concrete object (if present in the domain of the store), and ℓ^{*} (the *summary*) describes an unknown number of concrete objects. Typically, $\ell^{@}$ refers to the *most recently allocated* object from ℓ (hence the name of the technique), and ℓ^{*} refers to older objects – however the addition of interprocedural analysis (Section 7.4.3) changes this slightly.

In an intra-procedural setting, this mechanism is straightforward to incorporate. Informally, the transfer function for a node n of type $\mathsf{construct}[v]$ joins the $n^{@}$ object into the n^{*} object, redirects all pointers from $n^{@}$ to n^{*}, sets $n^{@}$ to an empty object, and assigns $n^{@}$ to v. Henceforth, v refers to a singleton abstract object, which permits strong updates.

The effect of incorporating recency abstraction on the analysis precision is substantial, as shown in Section 7.5.

7.4.3 Interprocedural Analysis

Function calls have a remarkably complicated semantics in JavaScript, but each step can be modeled precisely with our lattice definition. The transfer function for a call node n, $\mathsf{call}[w, v_0, \dots]$, extracts all function objects from w and then, as a side-effect, adds call edges to the entry nodes of these functions and return edges from their exit nodes back to the after-call node n' of n. To handle exception flow, return edges are also added from the exit-exc nodes to n'_{exc}, where n' has an exception edge to n'_{exc}. The call edge transfer function models parameter passing. It also models the new execution context being pushed onto the call stack. The base object, v_0, is used for setting `this` and the scope chain of the new execution context (which is why we need $\mathcal{P}(\mathsf{ScopeChain})$ in Obj).

A classical challenge in interprocedural analysis is to avoid flow through infeasible paths when a function is called from several sites [85]. Ignoring this effect may lead to a considerable loss of precision. We use the Modified component of Obj to keep track of object properties that may have been modified

since the current function was entered. For an abstract state σ_m at an exit node m with a return edge to an after-call node n', which belongs to a call node n, the edge transfer function checks whether the definitely non-modified parts of σ_m are inconsistent with σ_n, in which case it can safely discard the flow. (A given object property that is non-modified in σ_m is *consistent* with σ_n if its abstract value according to σ_n is less than or equal to its value according to σ_m.) If consistent, the transfer function replaces all non-modified parts of σ_m by the corresponding potentially more precise information from σ_n, together with the abstract stack. When propagating this flow along return edges, we must take into account the use of recency abstraction to "undo" the shuffling of singleton and summary objects. To this end, two sets of object labels are part of State to keep track of those object labels that are definitely/maybe summarized since entering the current function.

7.4.4 Termination of the Analysis

The usual termination requirement that the lattice should have finite height does not apply here, now even for a fixed program. We informally argue that the analysis nevertheless always terminates by the following observations: (1) The length of the ScopeChain object label sequences is always bounded by the lexical nesting depth of the program being analyzed. (2) The number of abstract states maintained for each node is solely determined by the choice of context sensitivity criteria. The simple heuristic proposed in Section 7.4 ensure the sizes of these sets to be bounded for any program. (3) The partial map in Obj has a potentially unbounded domain. However, at any point during fixpoint iteration a property name p can only occur in the domain if it was put in by a write-variable or write-property instruction. The property name for such an instruction comes from a temporary variable whose value is drawn from Value and coerced to String. In case that value is not a constant string, the use of *default_index* and *default_other* ensures that the domain is unmodified, and there are clearly only finitely many nodes that contain such an instruction. Together, these observations ensure that a fixpoint will be reached for any input program. The theoretical worst case complexity is obviously high, because of the complex analysis lattice. Nevertheless, our tool analyzes sizable programs within minutes, as shown in the next section.

7.5 Experiments

Our prototype is implemented on top of the JavaScript parser from Rhino [10] with around 17,000 lines of Java code. For testing that the prototype behaves as expected on the full JavaScript language, we have collected a corpus of more than 150 programs. These test programs are mostly in the range 5–50 lines of code and include 28 example programs[2] from Anderson et al. [3].

For the Anderson programs, our analysis detects all errors without spurious warnings and provides type information consistent with that of Anderson [3]. Our own programs were written to exercise various parts of the system and to provoke certain error messages, so it is not surprising that the analysis handles these well.

[2] `http://www.doc.ic.ac.uk/~cla97/js0impl/`

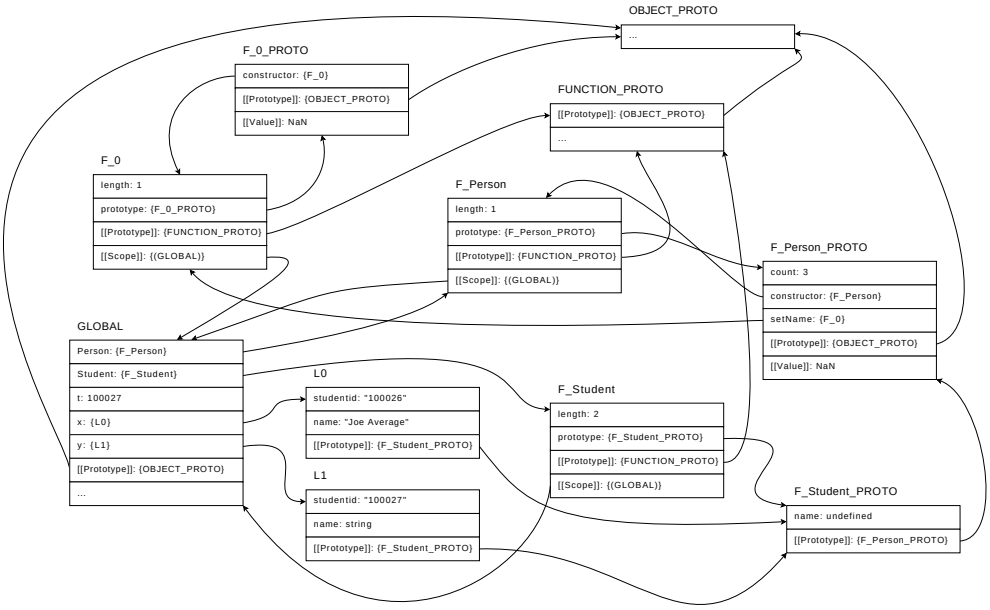


Figure 7.1: Abstract state for the final program point of the example program.

Running the analysis on the example program from Section 7.1 results in two warnings. First, the analysis correctly detects that the expression `s.toString()` involves a coercion from a primitive type to an object (which was deliberate by the programmer, in this case). Second, the analysis is able to prove that `y.studentid` is a string after the call to `y.setName`, but not that the string is a particular string, which results in a warning at the second `assert` statement. The reason is that `setName` is called twice on the same object with different strings (once through the constructor and once directly). A stronger heuristic for context sensitivity might resolve this issue.

Figure 7.1 shows the abstract state for the final program point of the example program, as obtained by running the prototype implementation. Each box describes an abstract object. For this simple program, each of them is a singleton (see Section 7.4.2). Edges correspond to references. For obvious reasons, only the used parts of the standard library are included in the illustration. The activation objects that are used during execution of the function calls have been removed by the abstract garbage collection. GLOBAL describes the global object, which also acts as execution context for the top-level code. OBJECT_PROTOTYPE and FUNCTION_PROTO model the prototype objects of the central built-in objects `Object` and `Function`, respectively. F_Person, F_Student, and F_0 correspond to the three functions defined in the program, and F_Person_PROTO, F_Student_PROTO, and F_0_PROTO are their prototype objects. Finally, L0 and L1 describe the two `Student` objects being created. The special property names [[Prototype]], [[Scope]], and [[Value]] are the so-called internal properties. For an example prototype chain, consider the object referred to by the variable `x` using the global object as variable object. Its prototype chain consists of L0, followed by F_Student_PROTO and

F_Person_PROTO, which reflects the sequence of objects relevant for resolving the expression `x.count`. As the illustration shows, even small JavaScript programs give rise to complex object structures, which our analysis lattice captures in sufficient detail.

The tool also outputs a call graph for the program in form of the call edges that are produced during fixpoint iteration, which can be useful for program comprehension.

The Google V8 benchmark suite[3] is our main testbed to evaluate the precision of the analysis on real code. It consists of four complex, standalone JavaScript programs. Although developed for testing performance of JavaScript interpreters, they are also highly demanding subjects for a static type analysis. In addition, we use the four most complex SunSpider benchmarks[4].

Clearly we do not expect to find bugs in such thoroughly tested programs, so instead we measure precision by counting the number of operations where the analysis does *not* produce a warning (for different categories), i.e. is capable of proving that the error cannot occur at that point.

For the `richards.js` benchmark (which simulates the task dispatcher of an operating system), the analysis shows for 95% of the 58 call/construct nodes that the value being invoked is always a function (i.e. category 1 from Section 7.1). Moreover, it detects one location where an absent variable is read (category 2). (In this case, the absent variable is used for feature detection in browsers.) This situation *definitely* occurs if that line is ever executed, and there are no spurious warnings for this category. Next, it shows for 93% of the 259 read/write/delete-property operations that they never attempt to coerce `null` or `undefined` into an object (category 3). For 87% of the 156 read-property operations where the property name is a constant string, the property is guaranteed to be present. As a bonus, the analysis correctly reports 6 functions to be dead, i.e. unreachable from program entry. We have not yet implemented checkers for the remaining categories of errors discussed in the introduction. In most cases, the false positives appear to be caused by the lack of path sensitivity.

The numbers for the `benchpress.js` benchmark (which is a collection of smaller benchmarks running in a test harness) are also encouraging: The analysis reports that 100% of the 119 call/construct operations always succeed without coercion errors, 0 warnings are reported about reading absent variables, 89% of the 113 read/write/delete-property operations have no coercion errors, and for 100% of the 48 read-property operations that have constant property names, the property being read is always present.

The third benchmark, `delta-blue.js` (a constraint solving algorithm), is larger and apparently more challenging for type analysis: 78% of the 182 call and construct instructions are guaranteed to succeed, 8 absent variables are correctly detected (all of them are functions that are defined in browser APIs, which we do not model), 82% of 492 read/write/delete-property instructions are proved safe, and 61% of 365 read-property with constant names are shown to be safe. For this benchmark, many of the false positives would likely be eliminated by better context sensitivity heuristics.

[3] `http://v8.googlecode.com/svn/data/benchmarks/v1/`

[4] `http://www2.webkit.org/perf/sunspider-0.9/sunspider.html`

	lines	call / construct	variable read	property access	fixed-property read
`richards.js`	529	95%	100%	93%	87%
`benchpress.js`	463	100%	100%	89%	100%
`delta-blue.js`	853	78%	100%	82%	61%
`3d-cube.js`	342	100%	100%	92%	100%
`3d-raytrace.js`	446	99%	100%	94%	94%
`crypto-md5.js`	291	100%	100%	100%	100%
`access-nbody.js`	174	100%	100%	93%	100%

Figure 7.2: Analysis precision.

The results for the first three V8 benchmarks and the four SunSpider benchmarks are summarized in Figure 7.2. For each of the categories discussed above, the table shows the ratio between precise answers obtained and the number of nodes of the relevant kind.

The fourth (and largest) V8 benchmark, `cryptobench.js`, presently causes our prototype to run out of memory (with a limit of 512MB). For the other benchmarks, analysis time is less than 10 seconds, except `3d-raytrace.js` and `delta-blue.js` which require 30 seconds and 6 minutes, respectively. Although analysis speed and memory consumption have not been key objectives for this prototype, we naturally pursue this matter further. Most likely, the work list ordering used by the fixpoint solver can be improved.

We can disable various features in the analysis to obtain a rough measure of their effect. Disabling abstract garbage collection has little consequence on the precision of the analysis on these programs, however it is cheap to apply and it generally reduces memory consumption. Using recency abstraction is crucial: With this technique disabled, the analysis of `richards.js` can only guarantee that a constant property is present in 2 of the 156 read-property nodes (i.e. less than 2%, compared to 87% before) and the number of warnings about potential dereferences of null or undefined rises from 19 to 90. These numbers confirm our hypothesis that recency abstraction is essential to the precision of the analysis. The Modified component of State is important for some benchmarks; for example, the number of warnings about dereferences of null or undefined in `3d-raytrace.js` rises from 21 to 61 if disabling this component. Finally, we observe that context sensitivity has a significant effect on e.g. `delta-blue.js`.

7.6 Conclusion

Scripting languages are a sweet-spot for applying static analysis techniques: There is yet little tool support for catching errors before code deployment and the programs are often relatively small. Our type analyzer is the first sound and detailed tool of this kind for real JavaScript code. The use of the monotone framework with an elaborate lattice structure, combined with recency abstraction, results in an analysis with good precision on demanding benchmarks.

We envision an IDE for JavaScript programming with features known from strongly typed languages, such as highlighting of type-related errors and sup-

port for precise content assists and safe refactorings. This goal requires further work, especially to improve the analysis speed. Our primary objectives for the prototype have been soundness and precision, so there are plenty of opportunities for improving performance. For example, we currently use a naive work list heuristic and the representation of abstract states employs little sharing.

In further experiments, we want to investigate if there is a need for even higher precision. For example, the String component could be replaced by regular languages obtained using a variant of string analysis [15]. It may also be fruitful to tune the context sensitivity heuristic or incorporate simple path sensitivity.

Another area is the consideration of the DOM, which is heavily used by most JavaScript programs. Our work provides a basis for modeling the different DOM implementations provided by the main browsers and hence for catching browser specific programming errors. Additionally, it paves the way for analyzing code that uses libraries (Dojo, Prototype, Yahoo! UI, FBJS, jQuery, etc.). With these further challenges ahead, the work presented here constitutes a starting point for developing precise and efficient program analysis techniques and tools that can detect errors (recall the list from Section 7.1) and provide type information for JavaScript programs used in modern Web applications.

Acknowledgments

We thank Julian Dolby and Stephen Fink for contributing the WALA framework to the research community, which helped us in the early phases of the project. Our work also benefited from inspiring discussions about JavaScript with Lars Bak and the Google Aarhus team.

Chapter 8

Interprocedural Analysis with Lazy Propagation

Abstract

We propose *lazy propagation* as a technique for flow- and context-sensitive interprocedural analysis of programs with objects and first-class functions where transfer functions may not be distributive. The technique is described formally as a systematic modification of a variant of the monotone framework and its theoretical properties are shown. It is implemented in a type analysis tool for JavaScript where it results in a significant improvement in performance.

8.1 Introduction

With the increasing use of object-oriented scripting languages, such as JavaScript, program analysis techniques are being developed as an aid to the programmers [29, 35, 93, 89, 5, 39]. Although programs written in such languages are often relatively small compared to typical programs in other languages, their highly dynamic nature poses difficulties to static analysis. In particular, JavaScript programs involve complex interplays between first-class functions, objects with modifiable prototype chains, and implicit type coercions that all must be carefully modeled to ensure sufficient precision.

While developing a program analysis for JavaScript [52] aiming to statically infer type information we encountered the following challenge: *How can we obtain a flow- and context-sensitive interprocedural dataflow analysis that accounts for mutable heap structures, supports objects and first-class functions, is amenable to non-distributive transfer functions, and is efficient and precise?* Various directions can be considered. First, one may attempt to apply the classical monotone framework [56] as a whole-program analysis with an iterative fixpoint algorithm, where function call and return flow is treated as any other dataflow. This approach turns out to be unacceptable: the fixpoint algorithm requires too many iterations, and precision may suffer because spurious dataflow appears via interprocedurally unrealizable paths. Another approach is to apply the IFDS technique [77], which eliminates those problems. However, it is restricted to distributive analyses, which makes it inapplicable in our situation. A further consideration is the functional approach [85] which models

each function in the program as a partial summary function that maps input dataflow facts to output dataflow facts and then uses this summary function whenever the function is called. However, with a dataflow lattice as large as in our case it becomes difficult to avoid reanalyzing each function a large number of times. Although there are numerous alternatives and variations of these approaches, we have been unable to find one in the literature that adequately addresses the challenge described above. Much effort has also been put into more specialized analyses, such as pointer analysis [41], however it is far from obvious how to generalize that work to our setting.

As an introductory example, consider this fragment of a JavaScript program:

```
function Person(n) { this.setName(n); }
Person.prototype.setName = function(n) { this.name = n; }
function Student(n,s) { Person.call(this, n);
                        this.studentid = s.toString(); }
Student.prototype = new Person;
var x = new Student("John Doe", 12345);
x.setName("John Q. Doe");
```

The code defines two "classes" with constructors `Person` and `Student`. `Person` has a method `setName` via its prototype object, and `Student` inherits `setName` and defines an additional field `studentid`. The `call` statement in `Student` invokes the super class constructor `Person`.

Analyzing the often intricate flow of control and data in such programs requires detailed modeling of points-to relations among objects and functions and of type coercion rules. TAJS is a whole-program analysis based on the monotone framework that follows this approach, and our first implementation is capable of analyzing complex properties of many JavaScript programs. However, our experiments have shown a considerable redundancy of computation during the analysis that causes simple functions to be analyzed a large number of times. If, for example, the `setName` method is called from other locations in the program, then the slightest change of any abstract state appearing at any call site of `setName` during the analysis would cause the method to be reanalyzed, even though the changes may be entirely irrelevant for that method. In this paper, we propose a technique for avoiding much of this redundancy while preserving, or even improving, the precision of the analysis. Although our main application is type analysis for JavaScript, we believe the technique is more generally applicable to analyses for object-oriented languages.

The main idea is to introduce a notion of "unknown" values for object fields that are not accessed within the current function. This prevents much irrelevant information from being propagated during the fixpoint computation. The analysis initially assumes that no fields are accessed when flow enters a function. When such an unknown value is read, a recovery operation is invoked to go back through the call graph and propagate the correct value. By avoiding to recover the same values repeatedly, the total amortized cost of recovery is never higher than that of the original analysis. With large abstract states, the mechanism makes a noticeable difference to the analysis performance.

Lazy propagation should not be confused with demand-driven analysis [45]. The goal of the latter is to compute the results of an analysis only at specific program points thereby avoiding the effort to compute a global result. In

contrast, lazy propagation computes a model of the state for each program point.

The contributions of this paper can be summarized as follows:

- We propose an ADT-based adaptation of the monotone framework to programming languages with mutable heap structures and first-class functions and exhibit some of its limitations regarding precision and performance.
- We describe a systematic modification of the framework that introduces *lazy propagation*. This novel technique propagates dataflow facts "by need" in an iterative fixpoint algorithm. We provide a formal description of the method to reason about its properties and to serve as a blueprint for an implementation.
- The lazy propagation technique is experimentally validated: It has been implemented into our type analysis for JavaScript, TAJS [52], resulting in a significant improvement in performance.

In the appendix we prove termination, relate lazy propagation with the basic framework—showing that precision does not decrease, and sketch a soundness proof of the analysis.

8.2 A Basic Analysis Framework

Our starting point is the classical monotone framework [56] tailored to programming languages with mutable heap structures and first-class functions. The mutable state consists of a heap of objects. Each object is a map from field names to values, and each value is either a reference to an object, a function, or some primitive value. Note that this section contains no new results, but it sets the stage for presenting our approach in Section 8.3.

8.2.1 Analysis Instances

Given a program Q, an instance of the monotone framework for an analysis of Q is a tuple $\mathcal{A} = (F, N, L, P, C, n_0, c_0, \mathsf{Base}, T)$ consisting of:

F: the set of *functions* in Q;

N: the set of *primitive statements* (also called *nodes*) in Q;

L: a set of *object labels* in Q;

P: a set of *field names* (also called *properties*) in Q;

C: a set of abstract *contexts*, which are used for context sensitivity;

$n_0 \in N$ and $c_0 \in C$: an initial statement and context describing the entry of Q;

Base: a *base lattice* for modeling primitive values, such as integers or booleans;

$T : C \times N \to \mathsf{AnalysisLattice} \to \mathsf{AnalysisLattice}$: a monotone *transfer function* for each primitive statement, where $\mathsf{AnalysisLattice}$ is a lattice derived from the above information as detailed in Section 8.2.2.

Each of the sets must be finite and the Base lattice must have finite height. The primitive statements are organized into intraprocedural control flow graphs [58],

and the set of object labels is typically determined by allocation-site abstraction [54, 13].

The notation $\mathit{fun}(n) \in F$ denotes the function that contains the statement $n \in N$, and $\mathit{entry}(f)$ and $\mathit{exit}(f)$ denote the unique entry statement and exit statement, respectively, of the function $f \in F$. For a function call statement $n \in N$, $\mathit{after}(n)$ denotes the statement being returned to after the call. A *location* is a pair (c, n) of a context $c \in C$ and a statement $n \in N$.

8.2.2 Derived Lattices

An analysis instance gives rise to a collection of derived lattices. In the following, each function space is ordered pointwise and each powerset is ordered by inclusion. For a lattice X, the symbols $\bot_X$, $\sqsubseteq_X$, and $\sqcup_X$ denote the bottom element (representing the absence of information), the partial order, and the least upper bound operator (for merging information). We omit the X subscript when it is clear from the context.

An *abstract value* is described by the lattice Value as a set of object labels, a set of functions, and an element from the base lattice:

$$\mathsf{Value} = \mathcal{P}(L) \times \mathcal{P}(F) \times \mathsf{Base}$$

An *abstract object* is a map from field names to abstract values:

$$\mathsf{Obj} = P \to \mathsf{Value}$$

An *abstract state* is a map from object labels to abstract objects:

$$\mathsf{State} = L \to \mathsf{Obj}$$

Call graphs are described by this powerset lattice:

$$\mathsf{CallGraph} = \mathcal{P}(C \times N \times C \times F)$$

In a call graph $g \in \mathsf{CallGraph}$, we interpret $(c_1, n_1, c_2, f_2) \in g$ as a potential function call from statement n_1 in context c_1 to function f_2 in context c_2.

Finally, an element of $\mathsf{AnalysisLattice}$ provides an abstract state for each context and primitive statement (in a forward analysis, the program point immediately *before* the statement), combined with a call graph:

$$\mathsf{AnalysisLattice} = (C \times N \ \to \ \mathsf{State}) \times \mathsf{CallGraph}$$

In practice, an analysis may involve additional lattice components such as an abstract stack or extra information associated with each abstract object or field omit such components to simplify the presentation as they are irrelevant to the features that we focus on here. Our previous paper [52] describes the full lattices used in our type analysis for JavaScript.

8.2.3 Computing the Solution

The *solution* to $\mathcal{A}$ is the least element $a \in \mathsf{AnalysisLattice}$ that solves these constraints:

$$\forall c \in C, n \in N : T(c, n)(a) \sqsubseteq a$$

$solve(\mathcal{A})$ **where** $\mathcal{A} = (F, N, L, P, C, n_0, c_0, \mathsf{Base}, T)$:
  $a := \bot_{\mathsf{AnalysisLattice}}$
  $W := \{(c_0, n_0)\}$
  while $W \neq \emptyset$ **do**
    select and remove (c, n) from W
    $T_a(c, n)$
  end while
  return a

Figure 8.1: The worklist algorithm. The worklist contains *locations*, i.e., pairs of a context and a statement. The operation $T_a(c, n)$ computes the transfer function for (c, n) on the current analysis lattice element a and updates a accordingly. Additionally, it may add new entries to the worklist W. The transfer function for the initial location (c_0, n_0) is responsible for creating the initial abstract state.

Computing the solution to the constraints involves fixpoint iteration of the transfer functions, which is typically implemented with a worklist algorithm as the one presented in Figure 8.1. The algorithm maintains a worklist $W \subseteq C \times N$ of locations where the abstract state has changed and thus the transfer function should be applied. Lattice elements representing functions, in particular $a \in \mathsf{AnalysisLattice}$, are generally considered as *mutable* and we use the notation $T_a(c, n)$ for the assignment $a := T(c, n)(a)$. As a side effect, the call to $T_a(c, n)$ is responsible for adding entries to the worklist W, as explained in Section 8.2.4. This slightly unconventional approach to describing fixpoint iteration simplifies the presentation in the subsequent sections.

Note that the solution consists of both the computed call graph and an abstract state for each location. We do not construct the call graph in a preliminary phase because the presence of first-class functions implies that dataflow facts and call graph information are mutually dependent (as evident from the example program in Section 8.1).

This fixpoint algorithm leaves two implementation choices: the order in which entries are removed from the worklist W, which can greatly affect the number of iterations needed to reach the fixpoint, and the representation of lattice elements, which can affect both time and memory usage. These choices are, however, not the focus of the present paper (see, e.g. [55, 58, 44, 6, 90]).

8.2.4 An Abstract Data Type for Transfer Functions

To precisely explain our modifications of the framework in the subsequent sections, we treat AnalysisLattice as an imperative ADT (abstract data type) [62] with the following operations:

- *getfield* : $C \times N \times L \times P \to \mathsf{Value}$
- *getcallgraph* : $() \to \mathsf{CallGraph}$
- *getstate* : $C \times N \to \mathsf{State}$
- *propagate* : $C \times N \times \mathsf{State} \to ()$
- *funentry* : $C \times N \times C \times F \times \mathsf{State} \to ()$
- *funexit* : $C \times N \times C \times F \times \mathsf{State} \to ()$

We let $a \in \mathsf{AnalysisLattice}$ denote the current, mutable analysis lattice element. The transfer functions can only access a through these operations.

The operation *getfield*(c, n, ℓ, p) returns the abstract value of the field p in the abstract object ℓ at the entry of the primitive statement n in context c. In the basic framework, *getfield* performs a simple lookup, without any side effects on the analysis lattice element:

a.*getfield*$(c \in C, n \in N, \ell \in L, p \in P)$:
  return $u(\ell)(p)$ **where** $(m, \_) = a$ **and** $u = m(c, n)$

The *getcallgraph* operation selects the call graph component of the analysis lattice element:

a.*getcallgraph*():
  return g **where** $(\_, g) = a$

Transfer functions typically use the *getcallgraph* operation in combination with the *funexit* operation explained below. Moreover, the *getcallgraph* operation plays a role in the extended framework presented in Section 8.3.

The *getstate* operation returns the abstract state at a given location:

a.*getstate*$(c \in C,\ n \in N)$:
  return $m(c, n)$ **where** $(m, \_) = a$

The transfer functions must not read the field values from the returned abstract state (for that, the *getfield* operation is to be used). They may construct parameters to the operations *propagate*, *funentry*, and *funexit* by updating a copy of the returned abstract state.

The transfer functions must use the operation *propagate*(c, n, s) to pass information from one location to another within the same function (excluding recursive function calls). As a side effect, *propagate* adds the location (c, n) to the worklist W if its abstract state has changed. In the basic framework, *propagate* is defined as follows:

a.*propagate*$(c \in C,\ n \in N,\ s \in \mathsf{State})$:
  let $(m, g) = a$
  if $s \not\sqsubseteq m(c, n)$ **then**
    $m(c, n) := m(c, n) \sqcup s$
    $W := W \cup \{(c, n)\}$
  end if

The operation *funentry*(c_1, n_1, c_2, f_2, s) models function calls in a forward analysis. It modifies the analysis lattice element a to reflect the possibility of a function call from a statement n_1 in context c_1 to a function entry statement *entry*(f_2) in context c_2 where s is the abstract state after parameter passing. (With languages where parameters are passed via the stack, which we ignore here, the lattice is augmented accordingly.) In the basic framework, *funentry* adds the call edge from (c_1, n_1) to (c_2, f_2) and propagates s into the abstract state at the function entry statement *entry*(f_2) in context c_2:

a.*funentry*$(c_1 \in C,\ n_1 \in N,\ c_2 \in C,\ f_2 \in F,\ s \in \mathsf{State})$:
  $g := g \cup \{(c_1, n_1, c_2, f_2)\}$ **where** $(\_, g) = a$
  a.*propagate*$(c_2, \mathit{entry}(f_2), s)$
  a.*funexit*$(c_1, n_1, c_2, f_2, m(c_2, \mathit{exit}(f_2)))$

Adding a new call edge also triggers a call to *funexit* to establish dataflow from the function exit to the successor of the new call site.

The operation $funexit(c_1, n_1, c_2, f_2, s)$ is used for modeling function returns. It modifies the analysis lattice element to reflect the dataflow of s from the exit of a function f_2 in callee context c_2 to the successor of the call statement n_1 with caller context c_1. The basic framework does so by propagating s into the abstract state at the latter location:

$a.funexit(c_1 \in C,\ n_1 \in N,\ c_2 \in C,\ f_2 \in F,\ s \in \mathsf{State})$:
  $a.propagate(c_1, after(n_1), s)$

The parameters c_2 and f_2 are not used in the basic framework; they will be used in Section 8.3. The transfer functions obtain the connections between callers and callees via the *getcallgraph* operation explained earlier. If using an augmented lattice where the call stack is also modeled, that component would naturally be handled differently by *funexit* simply by copying it from the call location (c_1, n_1), essentially as local variables are treated in, for example, IFDS [77].

This basic framework is sufficiently general as a foundation for many analyses for object-oriented programming languages, such as Java or C#, as well as for object-based scripting languages like JavaScript as explained in Section 8.4. At the same time, it is sufficiently simple to allow us to precisely demonstrate the problems we attack and our solution in the following sections.

8.2.5 Problems with the Basic Analysis Framework

The first implementation of TAJS, our program analysis for JavaScript, is based on the basic analysis framework. Our initial experiments showed, perhaps not surprisingly, that many simple functions in our benchmark programs were analyzed over and over again (even for the same calling contexts) until the fixpoint was reached.

For example, a function in the `richards.js` benchmark from the V8 collection was analyzed 18 times when new dataflow appeared at the function entry:

```
TaskControlBlock.prototype.markAsRunnable = function () {
  this.state = this.state | STATE_RUNNABLE;
};
```

Most of the time, the new dataflow had nothing to do with the `this` object or the `STATE_RUNNABLE` variable. Although this particular function body is very short, it still takes time and space to analyze it and similar situations were observed for more complex functions and in other benchmark programs.

In addition to this abundant redundancy, we observed – again not surprisingly – a significant amount of spurious dataflow resulting from interprocedurally invalid paths. For example, if the function above is called from two different locations, with the same calling context, their entire heap structures (that is, the State component in the lattice) become joined, thereby losing precision.

Another issue we noticed was time and space required for propagating the initial state, which consists of 161 objects in the case of JavaScript. These objects are mutable and the analysis must account for changes made to them by the program. Since the analysis is both flow- and context-sensitive, a typical element of AnalysisLattice carries a lot of information even for small programs.

Our first version of TAJS applied two techniques to address these issues: (1) Lattice elements were represented in memory using *copy-on-write* to make their constituents shared between different locations until modified. (2) The lattice was extended to incorporate a simple effect analysis called *maybe-modified*: For each object field, the analysis would keep track of whether the field might have been modified since entering the current function. At function exit, field values that were definitely not modified by the function would be replaced by the value from the call site. As a consequence, the flow of unmodified fields was not affected by function calls. Although these two techniques are quite effective, the lazy propagation approach that we introduce in the next section often supersedes the maybe-modified technique and renders copy-on-write essentially superfluous. In Section 8.4 we experimentally compare lazy propagation with both the basic framework and the basic framework extended with the copy-on-write and maybe-modified techniques.

8.3 Extending the Framework with Lazy Propagation

To remedy the shortcomings of the basic framework, we propose an extension that can help reducing the observed redundancy and the amount of information being propagated by the transfer functions. The key idea is to ensure that the fixpoint solver *propagates information "by need"*. The extension consists of a systematic modification of the ADT representing the analysis lattice. This modification implicitly changes the behavior of the transfer functions without touching their implementation.

8.3.1 Modifications of the Analysis Lattice

In short, we modify the analysis lattice as follows:

1. We introduce an additional abstract value, unknown. Intuitively, a field p of an object has this value in an abstract state associated with some location in a function f if the value of p is not known to be needed (that is, referenced) in f or in a function called from f.
2. Each call edge is augmented with an abstract state that captures the data flow along the edge after parameter passing, such that this information is readily available when resolving unknown field values.
3. A special abstract state, none, is added, for describing absent call edges and locations that may be unreachable from the program entry.

More formally, we modify three of the sub-lattices as follows:

$$\mathsf{Obj} = P \to \left(\mathsf{Value}{\downarrow_{\mathsf{unknown}}}\right)$$

$$\mathsf{CallGraph} = C \times N \times C \times F \to (\mathsf{State}{\downarrow_{\mathsf{none}}})$$

$$\mathsf{AnalysisLattice} = \left(C \times N \ \to \ (\mathsf{State}{\downarrow_{\mathsf{none}}})\right) \times \mathsf{CallGraph}$$

Here, $X{\downarrow_{\mathsf{y}}}$ means the lattice X lifted over a new bottom element y. In a call graph $g \in \mathsf{CallGraph}$ in the original lattice, the presence of an edge $(c_1, n_1, c_2, f_2) \in g$ is modeled by $g'(c_1, n_1, c_2, f_2) \neq \mathsf{none}$ for the corresponding call graph g' in the modified lattice. Notice that $\bot_{\mathsf{State}}$ is now the function that maps all object labels and field names to unknown, which is different from the element none.

```
a.getfield'(c ∈ C, n ∈ N, ℓ ∈ L, p ∈ P):
  if m(c,n) ≠ none where (m, _) = a then
    v := a.getfield(c,n,ℓ,p)
    if v = unknown then
      v := a.recover(c,n,ℓ,p)
    end if
    return v
  else
    return ⊥Value
  end if
```

Figure 8.2: Algorithm for $getfield'(c, n, \ell, p)$. This modified version of *getfield* invokes *recover* in case the desired field value is unknown. If the state is none according to a, the operation simply returns $\bot_{\mathsf{Value}}$.

8.3.2 Modifications of the Abstract Data Type Operations

Before we describe the systematic modifications of the ADT operations we motivate the need for an auxiliary operation, *recover*, on the ADT:

$$recover : C \times N \times L \times P \to \mathsf{Value}$$

Suppose that, during the fixpoint iteration, a transfer function $T_a(c, n)$ invokes $a.getfield(c, n, \ell, p)$ with the result unknown. This result indicates the situation that the field p of an abstract object ℓ is referenced at the location (c, n), but the field value has not yet been propagated to this location due to the lazy propagation. The *recover* operation can then compute the proper field value by performing a specialized fixpoint computation to propagate just that field value to (c, n). We explain in Section 8.3.3 how *recover* is defined.

The *getfield* operation is modified such that it invokes *recover* if the desired field value is unknown, as shown in Figure 8.2. The modification may break monotonicity of the transfer functions, however, as we argue in Appendix 8.7, the analysis still produces the correct result.

Similarly, the *propagate* operation needs to be modified to account for the lattice element none and for the situation where unknown is joined with an ordinary element. The latter is accomplished by using *recover* whenever this situation occurs. The resulting operation $propagate'$ is shown in Figure 8.3.

We then modify $funentry(c_1, n_1, c_2, f_2, s)$ such that the abstract state s is propagated "lazily" into the abstract state at the primitive statement $entry(f_2)$ in context c_2. Here, laziness means that every field value that, according to a, is not referenced within the function f_2 in context c_2 gets replaced by unknown in the abstract state. Additionally, the modified operation records the abstract state at the call edge as required in the modified CallGraph lattice. The resulting operation $funentry'$ is defined in Figure 8.4. (Without loss of generality, we assume that the statement at $exit(f_2)$ returns to the caller without modifying the state.) As consequence of the modification, unknown field values get introduced into the abstract states at function entries.

The *funexit* operation is modified such that every unknown field value appearing in the abstract state being returned is replaced by the corresponding field value from the call edge, as shown in Figure 8.5. In JavaScript, entering a

```
a.propagate'(c ∈ C, n ∈ N, s ∈ State):
  let (m, g) = a and u = m(c, n)
  s' := s
  if u ≠ none then
    for all ℓ ∈ L, p ∈ P do
      if u(ℓ)(p) = unknown ∧ s(ℓ)(p) ≠ unknown then
        u(ℓ)(p) := a.recover(c, n, ℓ, p)
      else if u(ℓ)(p) ≠ unknown ∧ s(ℓ)(p) = unknown then
        s'(ℓ)(p) := a.recover(c, n, ℓ, p)
      end if
    end for
  end if
  a.propagate(c, n, s')
```

Figure 8.3: Algorithm for $propagate'(c, n, s)$. This modified version of *propagate* takes into account that field values may be unknown in both a and s. Specifically, it uses *recover* to ensure that the invocation of *propagate* in the last line never computes the least upper bound of unknown and an ordinary field value. The treatment of unknown values in s assumes that s is recoverable with respect to the current location (c, n). If the abstract state at (c, n) is none (the least element), then that gets updated to s.

function body at a functions call affects the heap, which is the reason for using the state from the call edge rather than the state from the call statement. If we extended the lattice to also model the call stack, then that component would naturally be recovered from the call statement rather than the call edge.

Figure 8.6 illustrates the dataflow at function entries and exits as modeled by the $funexit'$ and $funentry'$ operations. The two nodes n_1 and n_2 represent function call statements that invoke the function f. Assume that the value of the field p in the abstract object ℓ, denoted $\ell.p$, is v_1 at n_1 and v_2 at n_2 where $v_1, v_2 \in$ Value. When dataflow first arrives at $entry(f)$ the $funentry'$ operation sets $\ell.p$ to unknown. Assuming that f does not access $\ell.p$ it remains unknown throughout f, so $funexit'$ can safely restore the original value v_1 by merging the state from $exit(f)$ with u_{g1} (the state recorded at the call edge) at $after(n_1)$. Similarly for the other call site, the value v_2 will be restored at $after(n_2)$. Thus, the dataflow for non-referenced fields respects the interprocedurally valid paths. This is in contrast to the basic framework where the value of $\ell.p$ would be $v_1 \sqcup v_2$ at both $after(n_1)$ and $after(n_2)$. Thereby, the modification of *funexit* may – perhaps surprisingly – cause the resulting analysis solution to be more precise than in the basic framework. If a statement in f writes a value v' to $\ell.p$ it will no longer be unknown, so v' will propagate to both $after(n_1)$ and $after(n_2)$. If the transfer function of a statement in f invokes $getfield'$ to obtain the value of $\ell.p$ while it is unknown, it will be recovered by considering the call edges into f, as explained in Section 8.3.3.

The *getstate* operation is not modified. A transfer function cannot notice the fact that the returned State elements may contain unknown field values, because it is not permitted to read a field value through such a state.

Finally, the *getcallgraph* operation requires a minor modification to ensure that its output has the same type although the underlying lattice has changed:

```
a.funentry'(c1 ∈ C, n1 ∈ N, c2 ∈ C, f2 ∈ F, s ∈ State):
  let (m, g) = a and u = m(c2, entry(f2))
  // update the call edge
  g(c1, n1, c2, f2) := g(c1, n1, c2, f2) ⊔ s
  // introduce unknown field values
  s' := ⊥State
  if u ≠ none then
    for all ℓ ∈ L, p ∈ P do
      if u(ℓ)(p) ≠ unknown then
        // the field has been referenced
        s'(ℓ)(p) := s(ℓ)(p)
      end if
    end for
  end if
  // propagate the resulting state into the function entry
  a.propagate'(c2, entry(f2), s')
  // propagate flow for the return edge, if any is known already
  let t = m(c2, exit(f2))
  if t ≠ none then
    a.funexit'(c1, n1, c2, f2, t)
  end if
```

Figure 8.4: Algorithm for $funentry'(c_1, n_1, c_2, f_2, s)$. This modified version of *funentry* "lazily" propagates s into the abstract state at $entry(f_2)$ in context c_2. The abstract state s' is unknown for all fields that have not yet been referenced by the function being called according to u (recall that $\bot_{\mathsf{State}}$ maps all fields to unknown).

```
a.getcallgraph'():
  return {(c1, n1, c2, f2) | g(c1, n1, c2, f2) ≠ none} where (_, g) = a
```

To demonstrate how the lazy propagation framework manages to avoid certain redundant computations, consider again the `markAsRunnable` function in Section 8.2.5. Suppose that the analysis first encounters a call to this function with some abstract state s. This call triggers the analysis of the function body, which accesses only a few object fields within s. The abstract state at the entry location of the function is unknown for all other fields. If new flow subsequently arrives via a call to the function with another abstract state s' where $s \sqsubseteq s'$, the introduction of unknown values ensures that the function body is only reanalyzed if s' differs from s at the few relevant fields that are not unknown.

8.3.3 Recovering Unknown Field Values

We now turn to the definition of the auxiliary operation *recover*. It gets invoked by *getfield'* and *propagate'* whenever an unknown element needs to be replaced by a proper field value. The operation returns the desired field value but also, as a side effect, modifies the relevant abstract states for function entry locations in a.

```
a.funexit'(c1 ∈ C, n1 ∈ N, c2 ∈ C, f2 ∈ F, s ∈ State):
  let (_, g) = a and u_g = g(c1, n1, c2, f2)
  s' := ⊥State
  for all ℓ ∈ L, p ∈ P do
    if s(ℓ)(p) = unknown then
      // the field has not been accessed, so restore its value from the call edge
      state
      s'(ℓ)(p) := u_g(ℓ)(p)
    else
      s'(ℓ)(p) := s(ℓ)(p)
    end if
  end for
  a.propagate'(c1, after(n1), s')
```

Figure 8.5: Algorithm for $funexit'(c_1, n_1, c_2, f_2, s)$. This modified version of *funexit* restores field values that have not been accessed within the function being called, using the value from before the call. It then propagates the resulting state as in the original operation.

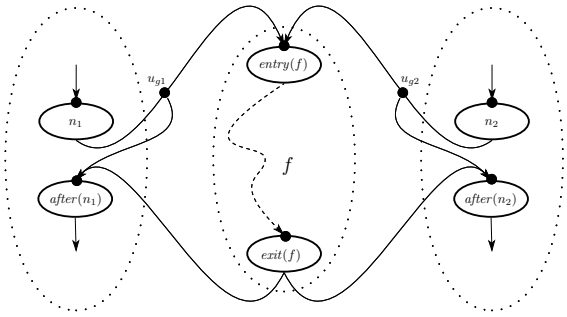


Figure 8.6: A function f being called from two different statements, n_1 and n_2 appearing in other functions (for simplicity, all with the same context c). The edges indicate dataflow, and each bullet corresponds to an element of State with $u_{g1} = g(c, n_1, c, f)$ and $u_{g2} = g(c, n_2, c, f)$ where $g \in \mathit{CallGraph}$.

The key observation for defining $recover(c, n, \ell, p)$ where $c \in C$, $n \in N$, $\ell \in L$, and $p \in P$ is that unknown is only introduced in $funentry'$ and that each call edge – very conveniently – records the abstract state just before the ordinary field value is changed into unknown. Thus, the operation needs to go back through the call graph and recover the missing information. However, it only needs to modify the abstract states that belong to function entry statements.

Recovery is a two phase process. The first phase constructs a directed multi-rooted graph G the nodes of which are a subset of $C \times F$. It is constructed from the call graph in a backward manner starting from (c, n) as the smallest graph satisfying the following two constraints, where $(m, g) = a$:

- The graph G contains the node $(c, fun(n))$.[1]
- For each node (c_2, f_2) in G and for each (c_1, n_1) where $g(c_1, n_1, c_2, f_2) \neq$ none:

[1] This constraint has been corrected after the SAS 2010 paper was published.

- If $u_g(\ell)(p) = \mathsf{unknown} \wedge u_1(\ell)(p) = \mathsf{unknown}$ where $u_g = g(c_1, n_1, c_2, f_2)$ and $u_1 = m(c_1, \mathit{entry}(\mathit{fun}(n_1)))$ then G contains the node $(c_1, \mathit{fun}(n_1))$ with an edge to (c_2, f_2),
- otherwise, (c_2, f_2) is a root of G.

The resulting graph is essentially a subgraph of the call graph. A node in G is a root if at least one of the incoming call graph edges of the corresponding function contributes with a non-unknown value. Notice that root nodes may have incoming edges in G.

The second phase is a fixpoint computation over G:

// recover the abstract value at the roots of G
for each root (c', f') of G **do**
  let $u' = m(c', \mathit{entry}(f'))$
  for all (c_1, n_1) **where** $g(c_1, n_1, c', f') \neq \mathsf{none}$ **do**
    let $u_g = g(c_1, n_1, c', f')$ **and** $u_1 = m(c_1, \mathit{entry}(\mathit{fun}(n_1)))$
    if $u_g(\ell)(p) \neq \mathsf{unknown}$ **then**
      $u'(\ell)(p) := u'(\ell)(p) \sqcup u_g(\ell)(p)$
    else if $u_1(\ell)(p) \neq \mathsf{unknown}$ **then**
      $u'(\ell)(p) := u'(\ell)(p) \sqcup u_1(\ell)(p)$
    end if
  end for
end for
// propagate throughout G at function entry nodes
$S :=$ the set of roots of G
while $S \neq \emptyset$ **do**
  select and remove (c', f') from S
  let $u' = m(c', \mathit{entry}(f'))$
  for each successor (c_2, f_2) of (c', f') in G **do**
    let $u_2 = m(c_2, \mathit{entry}(f_2))$
    if $u'(\ell)(p) \not\sqsubseteq u_2(\ell)(p)$ **then**
      $u_2(\ell)(p) := u_2(\ell)(p) \sqcup u'(\ell)(p)$
      add (c_2, f_2) to S
    end if
  end for
end while

This phase recovers the abstract value at the roots of G and then propagates the value through the nodes of G until a fixpoint is reached. Although *recover* modifies abstract states in this phase, it does not modify the worklist, an issue which we return to in Appendix 8.7.3. After this phase, we have $u(\ell)(p) \neq \mathsf{unknown}$ where $u = m(c', \mathit{entry}(f'))$ for each node (c', f') in G. (Notice that the side effects on a only concern abstract states at function entry statements.) In particular, this holds for $(c, \mathit{fun}(n))$, so when $\mathit{recover}(c, n, \ell, p)$ has completed the two phases, it returns the desired value $u(\ell)(p)$ where $u = m(c, \mathit{entry}(\mathit{fun}(n)))$.

Notice that the graph G is empty if $u(\ell)(p) \neq \mathsf{unknown}$ where $u = m(c, \mathit{entry}(\mathit{fun}(n)))$ (see the first of the two constraints defining G). In this case, the desired field has already been recovered, the second phase is effectively skipped, and $u(\ell)(p)$ is returned immediately.

Figure 8.7 illustrates an example of interprocedural dataflow among four functions. (This example ignores dataflow for function returns and assumes a fixed calling context c.) The statements write_1 and write_2 write to a field

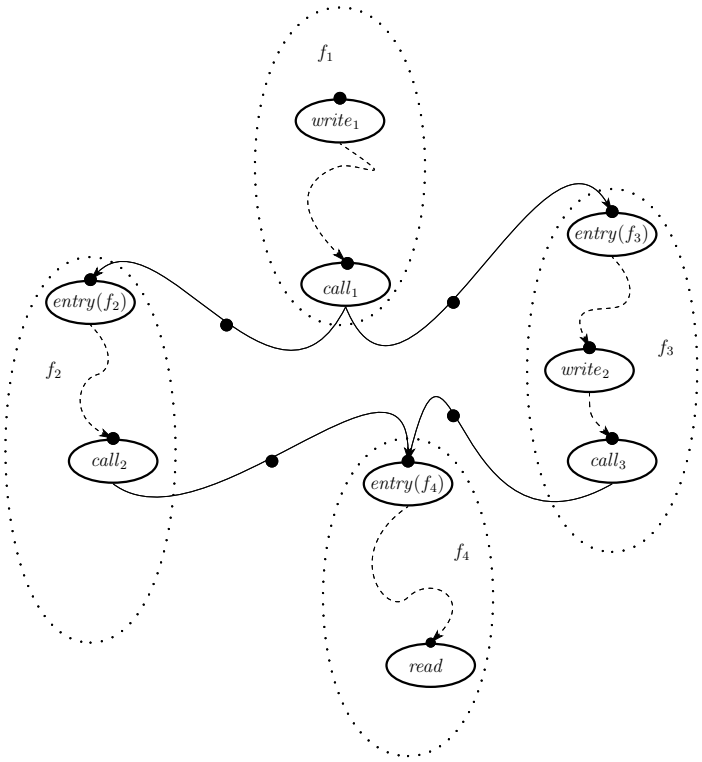


Figure 8.7: Fragments of four functions, $f_1 \ldots f_4$. As in Figure 8.6, edges indicate dataflow and bullets correspond to elements of State. The statements $write_1$ and $write_2$ write to a field $\ell.p$, and *read* reads from it. The *recover* operation applied to the *read* statement and $\ell.p$ will ensure that values written at $write_1$ and $write_2$ will be read at the *read* statements, despite the possible presence of unknown values.

$\ell.p$, and *read* reads from it. Assume that the analysis discovers all the call edges before visiting *read*. In that case, $\ell.p$ will have the value unknown when entering f_2 and f_3, which will propagate to f_4. The transfer function for *read* will then invoke *getfield*$'$, which in turn invokes *recover*. The graph G will be constructed with three nodes: (c, f_2), (c, f_3), and (c, f_4) where (c, f_2) and (c, f_3) are roots and have edges to (c, f_4). The second phase of *recover* will replace the unknown value of $\ell.p$ at $entry(f_2)$ and $entry(f_2)$ by its proper value stored at the call edges and then propagate that value to $entry(f_3)$ and finally return it to *getfield*$'$. Notice that the value of $\ell.p$ at, for example, the call edges, remains unknown. However, if dataflow subsequently arrives via transfer functions of other statements, those unknown values may be replaced by ordinary values. Finally, note that this simple example does not require fixpoint iteration within *recover*, however that becomes necessary when call graphs contain cycles (resulting from programs with recursive function calls).

The modifications only concern the AnalysisLattice ADT, in terms of which all transfer functions of an analysis are defined. The transfer functions them-

	LOC	Blocks	Iterations			Time (seconds)			Memory (MB)		
			basic	*basic+*	*lazy*	*basic*	*basic+*	*lazy*	*basic*	*basic+*	*lazy*
`richards.js`	529	478	2663	2782	1399	5.6	4.6	3.8	11.05	6.4	3.7
`benchpress.js`	463	710	18060	12581	5097	33.2	13.4	5.4	42.02	24.0	7.8
`delta-blue.js`	853	1054	∞	∞	63611	∞	∞	136.7	∞	∞	140.5
`cryptobench.js`	1736	2857	∞	43848	17213	∞	99.4	22.1	∞	127.9	42.8
`3d-cube.js`	342	545	7116	4147	2009	14.1	5.3	4.0	18.4	10.6	6.2
`3d-raytrace.js`	446	575	∞	30323	6749	∞	24.8	8.2	∞	16.7	10.1
`crypto-md5.js`	296	392	5358	1004	646	4.5	2.0	1.8	6.1	3.6	2.7
`access-nbody.js`	179	149	551	523	317	1.8	1.3	1.0	3.2	1.7	0.9

Table 8.1: Performance benchmark results.

selves are not changed. Although invocations of *recover* involve traversals of parts of the call graph, the main worklist algorithm (Figure 8.1) requires no modifications.

8.4 Implementation and Experiments

To examine the impact of lazy propagation on analysis performance, we extended the Java implementation of TAJS, our type analyzer for JavaScript [52], by systematically applying the modifications described in Section 8.3. As usual in dataflow analysis, primitive statements are grouped into basic blocks. The implementation focuses on the JavaScript language itself and the built-in library, but presently excludes the DOM API, so we use the most complex benchmarks from the V8[2] and SunSpider[3] benchmark collections for the experiments.

Descriptions of other aspects of TAJS not directly related to lazy propagation may be found in the TAJS paper [52]. These include the use of recency abstraction [7], which complicates the implementation, but does not change the properties of the lazy propagation technique.

We compare three versions of the analysis: *basic* corresponds to the basic framework described in Section 8.2; *basic+* extends the basic version with the copy-on-write and maybe-modified techniques discussed in Section 8.2.5, which is the version used in [52]; and *lazy* is the new implementation using lazy propagation (without the other extensions from the *basic+* version).

Table 9.1 shows for each program, the number of lines of code, the number of basic blocks, the number of fixpoint iterations for the worklist algorithm (Figure 8.1), analysis time (in seconds, running on a 3.2GHz PC), and memory consumption. We use ∞ to denote runs that require more than 512MB of memory.

We focus on the time and space requirements for these experiments. On our benchmark programs, the precision improvement is insignificant with respect to the number of potential type related bugs, which is the precision measure we have used in our previous work.

The experiments demonstrate that although the copy-on-write and maybe-modified techniques have a significant positive effect on the resource requirements, lazy propagation leads to even better results. The results for `richards.js`

[2] `http://v8.googlecode.com/svn/data/benchmarks/v1/`

[3] `http://www2.webkit.org/perf/sunspider-0.9/sunspider.html`

are a bit unusual as it takes more iterations in *basic+* than in *basic*, however the fixpoint is more precise in *basic+*.

The benchmark results demonstrate that lazy propagation results in a significant reduction of analysis time without sacrificing precision. Memory consumption is reduced by propagating less information during the fixpoint computation and fixpoints are reached in fewer iterations by eliminating a cause of redundant computation observed in the basic framework.

8.5 Related Work

Recently, JavaScript and other scripting languages have come into the focus of research on static program analysis, partly because of their challenging dynamic nature. These works range from analysis for security vulnerabilities [93, 35] to static type inference [29, 89, 3, 52]. We concentrate on the latter category, aiming to develop program analyses that can compensate for the lack of static type checking in these languages. The interplay of language features of JavaScript, including first-class functions, objects with modifiable prototype chains, and implicit type coercions, makes analysis a demanding task.

The IFDS framework by Reps, Horwitz, and Sagiv [77] is a powerful and widely used approach for obtaining precise interprocedural analyses. It requires the underlying lattice to be a powerset and the transfer functions to be distributive. Unfortunately, these requirements are not met by our type analysis problem for dynamic object-oriented scripting languages. The more general IDE framework also requires distributive transfer functions [81]. A connection to our approach is that fields that are marked as unknown at function exits, and hence have not been referenced within the function, are recovered from the call site in the same way local variables are treated in IFDS.

Sharir and Pnueli's functional approach to interprocedural analysis can be phrased both with symbolic representations and in an iterative style [85], where the latter is closer to our approach. With the complex lattices and transfer functions that appear to be necessary in analyses for object-oriented scripting languages, symbolic representations are difficult to work with, so TAJS uses the iterative style and a relatively direct representation of lattice elements. Furthermore, the functional approach is expensive if the analysis lattice is large.

Our analysis framework encompasses a general notion of context sensitivity through the C component of the analysis instances. Different instantiations of C lead to different kinds of context sensitivity, including variations of the call-string approach [85], which may also affect the quality of interprocedural analysis. We leave the choice of C open here; TAJS currently uses a heuristic that distinguishes call sites that have different values of `this`.

The use of unknown field values is related to the *maybe-modified* technique that we used in the first version of TAJS [52]: a field whose value is unknown is definitely not modified. Both ideas can be viewed as instances of side effect analysis. Unlike, for example, the side effect analysis by Landi et al. [80] our analysis computes the call graph on-the-fly and we exploit the information that certain fields are found to be non-referenced for obtaining the lazy propagation mechanism. Via this connection to side effect analysis, one may also view the unknown field values as establishing a frame condition as in separation logic [72].

Combining call graph construction with other analyses is common in pointer alias analysis with function pointers, for example in the work of Burke et al. [42]. That paper also describes an approach called deferred evaluation for increasing analysis efficiency, which is specialized to flow insensitive alias analysis.

Lazy propagation is related to lazy evaluation (e.g., [75]) as it produces values passed to functions on demand, but there are some differences. Lazy propagation does not defer evaluation as such, but just the propagation of the values; it applies not just to the parameters but to the entire state; and it restricts laziness to data structures (values of fields).

Lazy propagation is different from demand-driven analysis [45]. Both approaches defer computation, but demand-driven analysis only computes results for selected hot spots, whereas our goal is a whole-program analysis that infers information for all program points. Other techniques for reducing the amount of redundant computation in fixpoint solvers is difference propagation [25] and use of interprocedural def-use chains [90]. It might be possible to combine those techniques with lazy propagation, although they are difficult to apply to the complex transfer functions that we have in type analysis for JavaScript.

8.6 Conclusion

We have presented *lazy propagation* as a technique for improving the performance of interprocedural analysis in situations where existing methods, such as IFDS or the functional approach, do not apply. The technique is described by a systematic modification of a basic iterative framework. Through an implementation that performs type analysis for JavaScript we have demonstrated that it can significantly reduce the memory usage and the number of fixpoint iterations without sacrificing analysis precision. The result is a step toward sound, precise, and fast static analysis for object-oriented languages in general and scripting languages in particular.

Acknowledgments

The authors thank Stephen Fink, Michael Hind, and Thomas Reps for their inspiring comments on early versions of this paper.

8.7 Theoretical Properties

The lazy propagation analysis framework is supposed to improve on the results of the basic framework in several respects. First, we the modifications should not affect *termination*. Second, analysis results with lazy propagation should always be *at least as precise* as in the basic framework, meaning that the extensions introduce no spurious results. Third, the extensions should be *sound* in the sense that the analysis result is still a fixpoint of the transfer functions, which has to be adjusted because of the introduction of `unknown` field values, and that the transfer functions remain meaningful with respect to the language semantics. In the following, we state these properties more precisely and study them in some detail.

8.7.1 Termination

As observed in Section 8.3, the AnalysisLattice modifications do not preserve monotonicity of the transfer functions. Nevertheless, it is easy to see that the worklist algorithm (Figure 8.1) always terminates.

Proposition 1. *The worklist algorithm always terminates in the lazy propagation framework.*

Proof. Each AnalysisLattice operation terminates. The only nontrivial case is *recover*: Its first phase clearly terminates as only a finite set of nodes is considered, and the second phase terminates because AnalysisLattice has finite height.

Every iteration of the worklist algorithm removes a location from the worklist, and transfer functions only add new locations to the worklist when the lattice element is modified. As every such modification makes the lattice element larger and the lattice has finite height, termination is ensured. □

The number of iterations required to reach the fixpoint may differ due to the modifications. First, as mentioned in Section 8.2.3, we have left the worklist processing order unspecified and that order may be affected by the modifications. Second, as described in Section 8.3, the operation *funexit*′ improves precision with respect to the original *funexit* operation by avoiding certain interprocedurally invalid paths. Depending on the particular analysis instance, this improved precision may result in an increase or in a decrease of the number of iterations required to compute the fixpoint. In practice, we observe an overall decrease on each of our benchmark programs, as shown in Section 8.4.

The cost of performing a *recover* operation is proportional to the number of times it applies $\sqcup$. In the basic framework, the same amount of work is done, although "eagerly" within *propagate* operations. Hence, recovery does not impair the amortized analysis complexity.

8.7.2 Precision

For clarity, the text in this subsection marks all elements and lattices from the lazy propagation framework with primes $'$ whereas entities from the basic framework remain unadorned. Let $a_0 \in \mathsf{AnalysisLattice}$ be a solution of an analysis instance $\mathcal{A}$ in the basic framework, and let $a' \in \mathsf{AnalysisLattice}'$ be an intermediate step arising during the fixpoint iteration in the extended framework for $\mathcal{A}$. The goal is to show that a' is always smaller than a_0 in the lattice ordering, but this ordering cannot be directly established because the two lattices are different. Hence, we first need a function α that maps values of the extended analysis to values of the basic analysis. Figure 8.8 contains the definition of this function on the various lattices. It is easily seen to be bottom-preserving, monotone, and distributive.

The property that no spurious results arise with lazy propagation can now be stated as an invariant of the **while** loop in the worklist algorithm from Figure 8.1.

Proposition 2. *Let $\mathcal{A}$ be an analysis instance, $a_0 \in \mathsf{AnalysisLattice}$ be the solution of $\mathcal{A}$ in the basic framework, and $a' \in \mathsf{AnalysisLattice}'$ be the analysis*

$\alpha(m', g') = (\alpha(m'), \alpha(g'))$ where $(m', g') \in \mathsf{AnalysisLattice}'$
$\alpha(g') = \{x \in C \times N \times C \times F \mid g'(x) \neq \mathsf{none}\}$ where $g' \in \mathsf{CallGraph}'$
$\alpha(m')(c, n) = \alpha(m'(c, n))$ where $m' \in (C \times N \to \mathsf{State}'{\downarrow_{\mathsf{none}}}), c \in C, n \in N$
$\alpha(u')(\ell)(p) = \alpha(u'(\ell)(p))$ where $u' \in \mathsf{State}'{\downarrow_{\mathsf{none}}}, \ell \in L, p \in P,$ if $u' \neq \mathsf{none}$
$\alpha(\mathsf{none}) = \bot_{\mathsf{State}}$
$\alpha(v') = v'$ where $v' \in \mathsf{Value}{\downarrow_{\mathsf{unknown}}}$, if $v' \neq \mathsf{unknown}$
$\alpha(\mathsf{unknown}) = \bot_{\mathsf{Value}}$

Figure 8.8: Mapping between lattices in the extended and the basic framework.

lattice element on an entry to the **while** *loop in the worklist algorithm (Figure 8.1) applied to $\mathcal{A}$ with the lazy propagation framework. Then a' and a_0 are* α-related*, i.e., $\alpha(a') \sqsubseteq a_0$.*

Proof. On first entry to the loop, $a' = \bot_{\mathsf{AnalysisLattice}'}$. As α is bottom-preserving, $\alpha(a') \sqsubseteq a_0$. To establish the invariant, we assume that $\alpha(a') \sqsubseteq a_0$, let $t = T(c_0, n_0)$, for some $(c_0, n_0) \in C \times N$, and show that $\alpha(t(a')) \sqsubseteq a_0$.

As part of the computation of $t(a')$, the transfer function t may invoke the ADT operations on a', and we need to (1) check the effect of each operation on a' and prove that the α relation still holds. Additionally, since the output of one operation may be used as input to another and we may assume that the arguments of each invocation of an operation in a transfer function are computed by monotone functions, we are also obliged to (2) check that α-related arguments to the operations yield α-related results. In the following, we let $(m_0, g_0) = a_0$ and $(m', g') = a'$ and prove the properties (1) and (2) for each operation in turn.

Case *getcallgraph'*. The invocation of $a'.\mathit{getcallgraph}'()$ does not affect a'. The result is a subset of $a_0.\mathit{getcallgraph}()$ because $\alpha(a') \sqsubseteq a_0$.

Case *getstate*. This operation does not modify a'. For the result, we have $\alpha(a'.\mathit{getstate}(c, n)) \sqsubseteq a_0.\mathit{getstate}(c, n)$.

Case *getfield'*. Consider the invocation of $a'.\mathit{getfield}'(c, n, \ell, p)$. If $m'(c, n) = \mathsf{none}$, then a' is not changed and the result is $\bot$ which preserves the invariant. Let now $m'(c, n) \neq \mathsf{none}$ and $v = a'.\mathit{getfield}(c, n, \ell, p)$. If $v \neq \mathsf{unknown}$, then a' is not changed and $\alpha(v) \sqsubseteq a_0.\mathit{getfield}(c, n, \ell, p)$. If $v = \mathsf{unknown}$, then we need to consider the changes effected by *recover* where we also relate the result to the expected one.

Case *propagate'*. Consider the invocation of $a'.\mathit{propagate}'(c, n, s')$ from a transfer function $t = T(c_0, n_0)$, where (c_0, n_0) is a predecessor of (c, n). As a_0 is a solution, it holds that $t(a_0) \sqsubseteq a_0$ and that consequently $a_0.\mathit{propagate}(c, n, s)$ leaves a_0 unchanged, where $\alpha(s') \sqsubseteq s$ as both states are computed by the same monotone function from α-related arguments.

If $u' = m'(c, n)$ is none, then $m'(c, n)$ is effectively updated to s'. Now, $\alpha(m'(c, n)) = \alpha(s') \sqsubseteq s \sqsubseteq m(c, n)$ with the last equation holding because $a_0.\mathit{propagate}$ leaves a_0 unchanged.

Otherwise, parts of u' may need to be recovered which (assumedly) does not violate the invariant. We then have that $\alpha(m'(c, n)) \sqsubseteq m(c, n)$ before the invocation of *propagate* and $\alpha(m'(c, n) \sqcup s') = \alpha(m'(c, n)) \sqcup \alpha(s') \sqsubseteq m(c, n) \sqcup s \sqsubseteq m(c, n)$ afterwards.

This operation returns no result, so the α-relation trivially holds.

Case *recover*. Consider the invocation of $a'.recover(c, n, \ell, p)$. The first node added to the graph G is $(c, fun(n))$.

For this return value, it holds that $\alpha(v') \sqsubseteq m(c, entry(fun(n)))(\ell)(p)$ by assumption. By similar reasoning as in subcase B below, it must be that

$$m(c, entry(fun(n)))(\ell)(p) \sqsubseteq m(c, n)(\ell)(p) = a_0.getfield(c, n, \ell, p).$$

Hence, $\alpha(v') \sqsubseteq a_0.getfield(c, n, \ell, p)$ as required.

Once the graph G has been constructed, the recovery algorithm first examines the roots (c', f') of G and modifies their states in a'. Let (c', f') be such a root, $u' = m'(c', entry(f'))$, and let (c_1, n_1) be such that $u'_g = g'(c_1, n_1, c', f') \neq \mathsf{none}$. Let further $u'_c = m'(c_1, n_1)$ and $u'_1 = m'(c_1, entry(fun(n_1)))$.

As (c', f') is reachable there must have been a prior step in the fixpoint iteration where some transfer function $t' = T(c_1, n_1)$ invokes $funentry'$. Inside of this t' there must be a monotone function *invoke* which commutes with α and which constructs the State argument to $funentry'$ such that $u'_g = invoke(u'_c)$. This same function is also used in the verification that a_0 is a solution. In this verification, suppose that the State argument is $s = invoke(u_c)$ where $u_c = m_0(c_1, n_1)$. Let further $u = m_0(c', entry(f'))$ and $u_1 = m_0(c_1, entry(fun(n_1)))$.

Subcase A. Let us first assume that $u'_g(\ell)(p) \neq \mathsf{unknown}$. By our assumptions, it holds that $\alpha(u') \sqsubseteq u$ and $\alpha(u'_c) \sqsubseteq u_c$. Because $u'_g = invoke(u'_c)$ and $s = invoke(u_c)$ and *invoke* commutes with α, it also holds that $\alpha(u'_g) \sqsubseteq s$.

Now, let ${u'_g}^{\ell p}$ be bottom except at $\ell.p$ where it is equal to $u'_g(\ell)(p)$. With this setting, we can reason that

$$\begin{aligned}\alpha(u' \sqcup {u'_g}^{\ell p}) &\sqsubseteq \alpha(u' \sqcup u'_g) = \alpha(u' \sqcup invoke(u'_c)) \\ &= \alpha(u') \sqcup \alpha(invoke(u'_c)) \sqsubseteq u \sqcup invoke(u_c) = u\end{aligned}$$

where the last equality is due to the *propagate* operation in the standard *funentry* operation.

Subcase B. For the second case, assume that $u'_g(\ell)(p) = \mathsf{unknown}$ but $u'_1(\ell)(p) \neq \mathsf{unknown}$. As the algorithm propagates the latter value, we need to prove that it would not change if it were propagated to u'_c. In fact, to establish the invariant it is sufficient to show that $u_1(\ell)(p) \sqsubseteq u_c(\ell)(p)$ in the basic analysis.

Suppose for a contradiction that $u_1(\ell)(p) \not\sqsubseteq u_c(\ell)(p)$. Then there must be some n_x on a path between $n_e = entry(fun(n_1))$ and n_1 where each node between n_e and n_x satisfies $u_1(\ell)(p) \sqsubseteq m_0(c_1, n_e)(\ell)(p)$ but $u_1(\ell)(p) \not\sqsubseteq m_0(c_1, n_x)(\ell)(p)$. Let n'_x be the predecessor of n_x on this path. Clearly, $T(c_1, n'_x)$ changes the $\ell.p$ field by invoking $propagate(c_1, n'_x, s_x)$ for some $s_x = action(m_0(c_1, n'_x))$ with $s_x(\ell)(p) \sqsupset \bot$.

As the same transfer function must have been called in the extended framework (otherwise the function call at n_1 would not be reachable), there must have been an invocation of $propagate'(c_1, n'_x, s'_x)$ for some s'_x with $\alpha(s'_x) \sqsubseteq s_x$ and $s'_x(\ell)(p) \sqsupset \bot$ (because T never processes $\mathsf{unknown}$). But such an invocation contradicts $u'_g(\ell)(p) = \mathsf{unknown}$, so no such node n_x exists.

Hence, $\alpha(u'_1(\ell)(p)) \sqsubseteq u_1(\ell)(p) \sqsubseteq u_c(\ell)(p)$ so that

$$\begin{aligned}\alpha(u' \sqcup u'_1)(\ell)(p) &\sqsubseteq \alpha(u' \sqcup u'_1)(\ell)(p)\\ &= \alpha(u')(\ell)(p) \sqcup \alpha(u'_1)(\ell)(p)\\ &\sqsubseteq u(\ell)(p) \sqcup u_1(\ell)(p)\\ &\sqsubseteq u(\ell)(p) \sqcup u_c(\ell)(p)\\ &\sqsubseteq u(\ell)(p) \sqcup \mathit{invoke}(u_c)(\ell)(p)\\ &= u(\ell)(p)\end{aligned}$$

Thus, recovery at the roots does not violate the desired invariant. The final propagation does not do so either. It propagates state from the function entry node of the caller to the function entry node of the callee under the assumption that the corresponding component on the call edge is unknown. This assumption holds by construction of G. With the same argumentation as in the previous case, the state of the $\ell.p$ field cannot change between the entry to the caller and the actual call, so the invariant holds after each iteration of the loop and thus for the fixpoint as well.

The return value is extracted from $m'(c, \mathit{entry}(\mathit{fun}(n)))(\ell)(p)$ which α approximates the value $a_0.\mathit{getfield}(c, n, \ell, p)$ as explained in the beginning of this case.

Case *funentry′*. An invocation of $a'.\mathit{funentry}'(c_1, n_1, c_2, f_2, s')$ first adds s' to the call edge, which is correct because the corresponding call to $\mathit{funentry}(c_1, n_1, c_2, f_2, s)$ in the basic framework adds the tuple (c_1, n_1, c_2, f_2) to the basic call graph.

Next it computes a projection s'' of s', for which clearly $s'' \sqsubseteq s'$ and hence $\alpha(s'') \sqsubseteq s$ holds. With this precondition, the call to *propagate* preserves the invariant.

If the final call to *funexit′* does not happen, then there is no further change to a'. Otherwise, the invariant holds by assumption on *funexit*.

This operation returns no result, so again the α-relation trivially holds.

Case *funexit′*. Each invocation $a'.\mathit{funexit}'(c_1, n_1, c_2, f_2, s')$ happens with a state argument computed from the exit node of function f_2, such as, $n_2 = \mathit{exit}(f_2)$, so that $s' = f_{\mathit{exit}}(m'(c_2, n_2))$. Hence, the analogous call in the verification of the basic framework uses $s = f_{\mathit{exit}}(m(c_2, n_2))$, so that $\alpha(s') \sqsubseteq s$ holds, as usual.

Let furthermore $u'_g = g'(c_1, n_1, c_2, f_2)$ be the corresponding call edge and u_g the state parameter of the corresponding *funentry* call in the basic framework.

Let $LP = \{(\ell, p) \mid s'(\ell, p) = \mathsf{unknown}\}$. By similar reasoning as in the case for *recover*, for each $(\ell, p) \in LP$, it holds that $u_g(\ell)(p) \sqsubseteq m(c_2, n_2)(\ell)(p)$, that is, this state component is preserved from the invocation to the end of the function.

For the state s'' computed in *funexit′* we must argue that $\alpha(s'') \sqsubseteq s$ which is not obvious. For $(\ell, p) \notin LP$, it holds that $\alpha(s''(\ell)(p)) = \alpha(s'(\ell)(p)) \sqsubseteq s(\ell)(p)$ by assumption $\alpha(s') \sqsubseteq s$. For $(\ell, p) \in LP$, it holds that $\alpha(s''(\ell)(p)) = \alpha(u'_g(\ell)(p)) \sqsubseteq u_g(\ell)(p) \sqsubseteq m(c_2, n_2)(\ell)(p) = s(\ell)(p)$.

Hence, the final call to *propagate′* happens with α-related arguments and does not destroy the invariant.

This operation returns no result, so again the α-relation trivially holds. □

8.7.3 Soundness

The changes made to the AnalysisLattice operations indirectly modify the transfer functions, so it is also important that these remain sound with respect to the semantics of the program. To state this more precisely, let $[\![Q]\!]$ be a collecting semantics of a program Q (in the abstract interpretation sense [17]) such that $\beta_{[\![Q]\!]}$ is an abstraction of $[\![Q]\!]$ in the domain AnalysisLattice from Section 8.2 expressed via the operations *getfield* and *getcallgraph*. We say that $a \in$ AnalysisLattice (using either the basic framework or lazy propagation) over-approximates $\beta_{[\![Q]\!]}$ if

$$\beta_{[\![Q]\!]}.getfield \sqsubseteq a.getfield \ \wedge \ \beta_{[\![Q]\!]}.getcallgraph \sqsubseteq a.getcallgraph$$

We conjecture that lazy propagation is then sound in the following sense:

Assume that $a_0 \in$ AnalysisLattice is the solution in the basic analysis framework of an analysis instance $\mathcal{A}$ for a program Q and that a_0 over-approximates $\beta_{[\![Q]\!]}$. If a_0' is the solution of $\mathcal{A}$ in the lazy propagation framework then a_0' also over-approximates $\beta_{[\![Q]\!]}$.

Without giving a full proof, we mention some key aspects of the reasoning. Most importantly, lazy propagation gives a safe approximation compared to the maybe-modified technique briefly mentioned in Section 8.2.5, and that technique is clearly sound relative to the basic framework.

The worklist algorithm for the basic framework produces a solution to the analysis in the sense defined in Section 8.2.3. A requirement for this to hold is that every AnalysisLattice ADT operation that modifies an abstract state at some location also adds that location to the worklist. This requirement is also fulfilled with lazy propagation – except for a subtlety in the *recover* operation: It modifies states that belong to function entry locations without adding these to the worklist. This means that such values that have been recovered at the function entry locations may not be propagated. However, recall that transfer functions can only read object field values via the *getfield'* operation. Assume that $getfield'(c, n, \ell, p)$ is invoked and the field $\ell.p$ is unknown at the location (c, n). In that case, *getfield'* will call *recover*, and in the situation where the proper value v has already been recovered at the function entry location $(c, entry(fun(n)))$ the value v is returned by *getfield'*. This means that the transfer function will behave in the same way as if v had been propagated from the function entry location. A similar situation occurs if the recovery has taken place not at the same function but at an earlier location in the call graph. Thus, the fact that *recover* modifies abstract states without adding their locations to the worklist does not affect correctness of the analysis result.

Chapter 9

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

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.

9.1 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 10.1 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 [52, 51]. To this point, TAJS has been developed to faithfully model the JavaScript language and the core library as specified in the ECMAScript standard [23]. 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 [79]. We now take the step of extending TAJS to also model these aspects of JavaScript

```
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 10.6.2.

Figure 9.1: Excerpts from the Google Chrome Experiment *JS Touch*[1].

web applications.

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

- We discuss the key challenges (Section 9.2) and suggest an approach toward modeling the JavaScript web application platform in static analysis (Section 9.4). In particular, this involves considerations about modeling the HTML pages and the event system.
- We show how the TAJS analysis (Section 9.2.2) 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 9.5). We identify strengths and weaknesses of the approaches we have taken and suggest directions for future work (Section 9.7).

Several program analysis tools and techniques for JavaScript have been developed [89, 3, 49, 27, 35, 63, 37, 16, 33, 32, 52], 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 9.6.

9.2 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.

9.2.1 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. [79], 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 [23], 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. [64]. One choice we must

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

make is whether to model the standard or one or more of the existing implementations. We return to this issue in Section 9.3.

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 [59, 79, 78].

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.

9.2.2 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 nonstandard 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 <script type=''text/javascript''>
2   var src = "foo.png";
3 </script>
4 <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

[2] `http://www.w3.org/DOM/`
[3] `http://www.whatwg.org/`
[4] `http://www.quirksmode.org/`
[5] `http://www.chromeexperiments.com/detail/domtris/`

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.

9.2.3 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[6] tool for collecting visitor statistics:

```
1 <script type="text/javascript">
2   (function() {
3     var ga = document.createElement('script');
4     ga.type = 'text/javascript';
5     ga.async = true;
6     ga.src =
7       ('https:' == document.location.protocol ?
8        'https://ssl' :
```

[6] http://www.google.com/analytics/

```
9         'http://www') + '.google-analytics.com/ga.js';
10      var s = document.getElementsByTagName('script')[0];
11      s.parentNode.insertBefore(ga, s);
12    })();
13  </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.

9.3 The TAJS Analyzer

We base the current work on the TAJS analysis tool that is described in previous publications [52, 51]. TAJS is a whole-program flow analysis that supports the full JavaScript language as defined in the ECMA-262 specification [23], 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 9.2.1, 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 [56] 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 [7].

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 [51].

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

9.4 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`[7] 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.

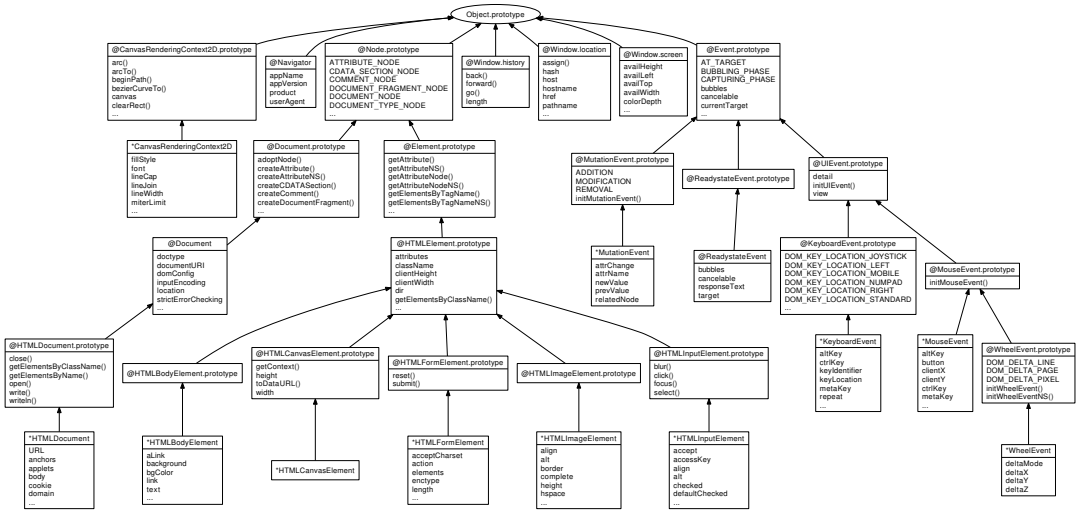


Figure 9.2: An excerpt from the HTML object hierarchy.

In total, the extensions comprise around 250 abstract objects with 500 properties and 200 transfer functions. To give an impression of the complexity, Figure 9.2 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.

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

9.4.1 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 [33, 35] 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 9.2) 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 9.2.2), 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.

9.4.2 Events

As discussed in Section 9.2.2, 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 9.5.

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`

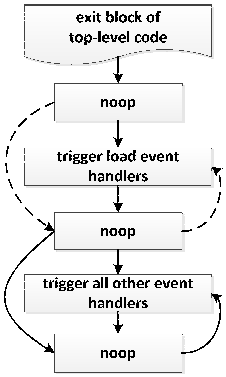


Figure 9.3: Modeling events in the flow graphs.

or assignments to event attributes during 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 [52]). 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 9.3. (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.

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 9.2) 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 9.2.2) into account.

9.4.3 Special Object Properties

As discussed in Section 9.2.2, 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:

```
1 var i = document.createElement("img");
2 f.id = "myImage";
3 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 9.5), this occurs 0 times, indicating that the analysis is generally precise enough to avoid the problem.

9.4.4 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*[8]:

```
1 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.

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

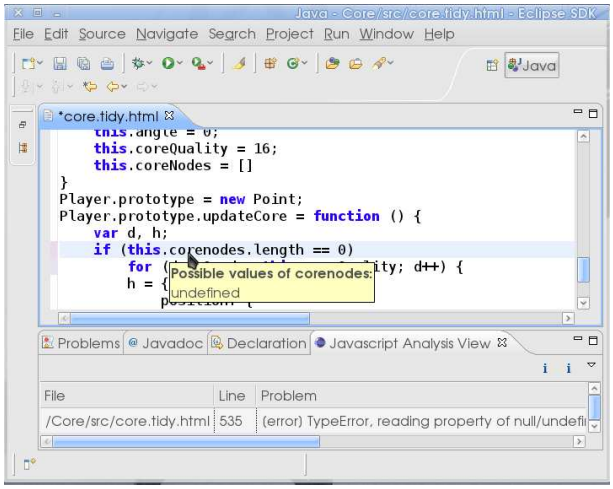


Figure 9.4: The TAJS analysis plug-in for Eclipse, reporting a programming error and highlighting the type inferred for the selected expression.

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.

9.5 Evaluation

We have extended the pre-existing TAJS analysis tool according to Section 9.4. The tool is implemented in Java and uses the JavaScript parser from the Mozilla Rhino project[9]. 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 9.4.

9.5.1 Research Questions

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

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

Q1 We wish to study the ability of the tool to detect programming errors of the kinds discussed in Section 9.1. 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 9.4.3), 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 9.1), 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 unreachable 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.

9.5.2 Benchmark Programs

Our benchmark programs are drawn from three different sources: *Chrome Experiments*[10], *Internet Explorer 9 Test Drive*[11] and the *10K Apart Challenge*[12]. 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 9.4.4. Moreover, analyzing applications that involve large libraries, such as jQuery, MooTools, and Prototype, is particularly challenging for the reasons discussed in Section 9.2.3. At present, we limit our level of ambition to applications that do not depend on such libraries. The applications

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

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 9.1 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 9.4.4 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.

9.5.3 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).

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 9.1, 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

	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 9.1: 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).

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 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 9.1 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 9.1 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 9.1 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 9.4.3), 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 9.5.2 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.

9.6 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 [3]. The abstract domain used in TAJS subsumes such information. Other early work includes Thiemann's type system [89]. 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 [88].

More recently, Jang and Choe have presented a points-to analysis for a restricted subset of JavaScript based on set constraints [49]. 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 [33, 32]. 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 [35]. 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 [37]. 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 [16]. 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 9.2.

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 [63]. 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 [79]. 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 [78]. 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 9.2.3, is outside the scope of TAJS.

9.7 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 anal-

ysis 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.

Chapter 10

Remedying the Eval that Men Do

Abstract

A range of static analysis tools and techniques have been developed in recent years with the aim of helping JavaScript web application programmers produce code that is more robust, safe, and efficient. However, as shown in a previous large-scale study, many web applications use the JavaScript `eval` function to dynamically construct code from text strings in ways that obstruct existing static analyses. As a consequence, the analyses either fail to reason about the web applications or produce unsound or useless results.

We present an approach to soundly and automatically transform many common uses of `eval` into other language constructs to enable sound static analysis of web applications. By eliminating calls to `eval`, we expand the applicability of static analysis for JavaScript web applications in general.

The transformation we propose works by incorporating a refactoring technique into a dataflow analyzer. We report on our experimental results with a small collection of programming patterns extracted from popular web sites. Although there are inevitably cases where the transformation must give up, our technique succeeds in eliminating many nontrivial occurrences of `eval`.

10.1 Introduction

The `eval` function and its variants in JavaScript allow dynamic construction of code from text strings. This can be useful for parsing JSON data[1], lazy loading of code[2], and execution of code provided by users in web-based JavaScript IDEs[3]. Using `eval`, however, makes it difficult to statically reason about the behavior of the application code. Existing automated static analyses for JavaScript try to dodge this problem. They either forbid `eval` altogether [89, 3, 49, 63], handle only the simplest cases where the strings passed to `eval` are constants or assumed to contain JSON data [52, 50, 35], or simply ignore all calls to `eval` thereby sacrificing precision and soundness [34]. Since JavaScript has limited encapsulation mechanisms, the dynamically constructed code can

[1] `http://www.json.org/js.html`
[2] `http://ajaxpatterns.org/On-Demand_Javascript`
[3] `http://tide4javascript.com/`

generally affect most of the application state, so ignoring calls to `eval` may have drastic consequences for the analysis quality.

The recommended best practice for web application developers is to avoid `eval`: *"The `eval` function is the most misused feature of JavaScript. Avoid it."* [18]. Nevertheless, the recent study "The Eval That Men Do" by Richards et al. has shown that `eval` is widely used [78]. Not only do a majority of the most popular web sites use `eval`, but in many cases they use it where simple alternatives exist, for example to access variables in the global scope or to access properties of objects. A likely explanation is poor understanding of the JavaScript language, in particular of its functional programming features that allow functions to be passed as arguments and of its unusual object model where each object is effectively a map from strings to values. Consequently, there is currently a mismatch between the capabilities of state-of-the-art static analysis tools for JavaScript and the JavaScript code that average programmers write.

Richards et al. also suggest that many of the uses of `eval` could be eliminated by rewriting the code, often improving both clarity and robustness as a side effect. They conclude that 83% of `eval` uses in their study could be rewritten to use less dynamic language features – however, they provide no automated way to perform these changes. Although it is often "obvious" to competent programmers how specific calls to `eval` can be eliminated manually, automating the transformation is not trivial. On the other hand, not all occurrences of `eval` can be eliminated with reasonable means; as an example, a call to `eval` that gets its input from an HTML text field could ultimately be eliminated by implementing a full JavaScript interpreter in JavaScript, which would hardly help static analysis tools reason about the code.

The goal of our work is to develop a sound, automated transformation technique for eliminating typical patterns of `eval` calls in JavaScript programs. The primary purpose is not to clean up messy code but rather to enable static analysis of programs that contain `eval`, for example for verification or bug detection. We therefore accept transformations that produce complex code as output as long as that code – unlike the input code that uses `eval` – is amenable to static analysis. We only permit transformations that preserve the behavior of the code because we want to apply sound static analyses on the resulting code. In this way, eliminating `eval` can be viewed as a code refactoring challenge [26]. We want a tool to transform the program code without affecting its behavior, which requires an analysis to check certain preconditions and infer other information needed by the transformation. This apparently raises a chicken-and-egg problem: Before we can rewrite a given occurrence of `eval` we need to run a static analysis to infer the necessary information, but as discussed above we cannot in general perform static analysis of programs that use `eval`.

Another challenge is that the flexibility of `eval` makes apparently simple cases surprisingly difficult. For example, consider a rewrite rule that replaces a call `eval("`S`")` by S when S is a constant string consisting of syntactically correct JavaScript code. Such a rule is unsound; for example, S may contain variable and function declarations even when the call `eval("`S`")` occurs inside an expression, so the resulting code might not be syntactically correct, and moreover, variable declarations in S may conflict with variables in the surrounding code. Even finding the occurrences of calls to `eval` is nontrivial because programs may create aliases of the `eval` function. Some programs use

```
1  function _var_exists(name) {
2    // return true if var exists in "global" context,
3    // false otherwise
4    try {
5      eval('var foo = ' + name + ';');
6    }
7    catch (e) {
8      return false;
9    }
10   return true;
11 }
12 var Namespace = {
13   // simple namespace support for classes
14   create: function(path) {
15     // create namespace for class
16     var container = null;
17     while (path.match(/^(\w+)\.?/)) {
18       var key = RegExp.$1;
19       path = path.replace(/^(\w+)\.?/, "");
20       if (!container) {
21         if (!_var_exists(key))
22           eval('window.' + key + ' = {};');
23         eval('container = ' + key + ';');
24       }
25       else {
26         if (!container[key]) container[key] = {};
27         container = container[key];
28       }
29     }
30   }
31 };
```

Figure 10.1: Example of `eval` taken from the Chrome Experiments program `canvas-cycle`.

such aliasing to exploit a subtlety of the language specification: Calling `eval` directly will cause the given code to be executed in the current scope, whereas calls via aliases use the global scope (or, before 5th edition of ECMAScript, cause an `EvalError` exception).

The example in Figure 10.1 demonstrates how `eval` can be used in practice. The code appears in the Chrome Experiments program `canvas-cycle`[4] and is part of a larger library that implements a class system in JavaScript, which does not support classes natively. This particular snippet implements a namespace mechanism for these classes.

The example contains three calls to `eval`. The first on line 5 tests whether a given name exists in the global scope (although it only works if the name is not `"name"` or `"foo"`). This could have been accomplished without `eval`, for example by writing `name in window` since `window` refers to the global scope. The second call to `eval` on line 22 is used to assign to a dynamically computed property of the `window` object. This could have been achieved using `window[key]` to access the dynamically computed property. The last `eval` call on line 23 could be rewritten in a similar way. This example demonstrates that many calls to `eval` are in fact unnecessary and the same results could be achieved with other language constructs that are easier to reason about for a static analysis.

[4] `http://www.chromeexperiments.com/detail/canvas-cycle/`

10.1.1 Contributions

The key idea of our approach is to eliminate `eval` calls soundly and automatically by incorporating refactoring into the fixpoint computation of a dataflow analyzer. We demonstrate this idea using the TAJS analysis [52, 51, 50] that performs a whole-program dataflow analysis for JavaScript web applications, but until now with poor support for `eval`. Whenever the analysis encounters dataflow into `eval`, a refactoring component is triggered for rewriting the call to equivalent JavaScript code without the `eval` call, and the analysis can proceed by analyzing the resulting code. When the analysis reaches its fixpoint, we have eliminated all reachable calls to `eval` and can output the resulting program. The success of this approach naturally depends on the power of the refactoring component and the information it can obtain from the underlying dataflow analysis – especially information about the strings that are passed to `eval`.

As an example, consider this fragment of JavaScript code used by Richards et al. for illustrating the power of `eval` [78]:

```
1 Point = function() {
2   var x=0; var y=0;
3   return function(o,f,v) {
4     if (o=="r")
5        return eval(f);
6     else
7        return eval(f+"="+v);
8     }
9 }
```

A call `p = Point()` will return a closure that can be invoked as e.g. `p("w", "x", 42)` to write the value `42` to the local variable `x` or as `p("r", "x")` to read its current value. Let us focus on the second `eval` call site. Suppose that our dataflow analysis first encounters a call `p("w", "x", 42)`. Provided that the analysis can keep track of the flow of values, it can infer that `eval` is called with the argument `"x"+"="+42`, which reduces to `eval("x=42")`. This `eval` call can safely be rewritten to the assignment `x=42`, and the dataflow analysis can proceed by analyzing the effect of that assignment, which will likely have consequences to other parts of the program. If the analysis later encounters another call, for example `p("w", "y", 87)`, things become more complicated. Even if the analysis knows that the value of `f` is always a valid, non-reserved identifier name and `v` is always a number, and the local variables `x` and `y` are merely properties of a scope object, it is difficult to rewrite the `eval`'d assignment `f+"="+v` into an object property assignment because JavaScript does not provide a way to obtain a reference to the local scope object. However, a context sensitive dataflow analysis can keep the two calls to `p` apart. Assuming that the analysis in this way finds out that the only possible values of `f` are `"x"` and `"y"`, the code may safely be transformed into the following by conditionally specializing the `eval` calls accordingly:

```
1 Point = function() {
2   var x=0; var y=0;
3   return function(o,f,v) {
4     if (o=="r")
5        return f==="x" ? x : y;
6     else
7        return f==="x" ? x=v : y=v;
```

```
8       }
9   }
```

Another example is the function `get_server_option` in the code for the web site `scribd.com`:

```
var get_server_option =
  function(name, default_value) {
  if (typeof Scribd.ServerOptions == 'undefined' ||
      eval('typeof Scribd.ServerOptions.' + name)
        == 'undefined')
    return default_value;
  return eval('Scribd.ServerOptions.' + name);
};
```

The dataflow analysis can find out that the value of `name` is always a valid identifier name by looking at the call sites, so the code can safely be rewritten to eliminate the calls to `eval`:

```
var get_server_option =
  function(name, default_value) {
  if (typeof Scribd.ServerOptions == 'undefined' ||
      typeof Scribd.ServerOptions[name]
        == 'undefined')
    return default_value;
  return Scribd.ServerOptions[name];
};
```

The transformations in these examples allow subsequent program analyses to reason about the code without having to worry about `eval`.

This paper explores the idea of incorporating `eval` refactoring into the dataflow analysis fixpoint computation and proposes a sequence of steps for developing the refactoring component and exploiting information provided by the dataflow analysis. In summary, the contributions of this paper are as follows.

- We describe a framework that soundly integrates refactoring of `eval` calls into a dataflow analyzer.
- Guided by a study of how `eval` is being used in practice, we instantiate our framework with different techniques for transforming typical calls to `eval` into equivalent JavaScript code without `eval`.
- We present results of an experimental evaluation with a prototype implementation. On 28 nontrivial programming patterns extracted from the Alexa top 500 web sites and from Chrome Experiments[5] containing a total of 44 calls to `eval`, our approach successfully eliminates 33 of the calls, which enables further use of static analysis on those applications and demonstrates that our approach is feasible. For the other call sites, we describe the challenges that remain for future work.

The remainder of this paper is organized as follows. Section 10.2 contains a study of calls to `eval`, slightly extending the work by Richards et al., to learn more about how `eval` is being used in practice. We present an overview of our transformation framework, the *Unevalizer*, in Section 10.3. We take the first step in Section 10.4 to eliminate a class of calls to `eval` where the arguments are constant strings and proceed with a number of improvements in Section 10.5 involving constant propagation, special treatment of strings that

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

contain JSON data or identifiers, and context sensitive specialization to obtain more precise information about the strings that enter `eval`. In Section 10.6 we report on experiments performed using our prototype implementation on a small collection of JavaScript web applications that use `eval` and until now have been out of reach for static analysis tools.

Although our presentation focuses on the `eval` function, our technique also works for its cousins `Function`, `setInterval`, and `setTimeout`, and in principle for script code embedded in dynamically constructed HTML and CSS data. We target the 3rd edition of ECMAScript [23]. None of the web sites we have studied use the newer strict mode semantics in combination with `eval`.

The intended user of our code transformation tool is the JavaScript web application developer. This means that we can disregard "minification" and lazy code loading, which are often used before deployment to compress the code and divide it into small parts for faster loading, and we can assume that all relevant source files are available for analysis.

We strive toward transformations that preserve the program behavior: Given a program that uses `eval`, our tool either outputs a program with the same external behavior, but without `eval`, or the tool gives up and issues an explanation message. (Stating this formally and proving correctness is beyond the scope of the paper.) Since the main purpose of our work is to enable sound static analysis of programs that use `eval`, one may argue that we could loosen this requirement and permit non-behavior preserving transformations as long as they are sound with respect to the subsequent analysis. The advantage of our present approach is that the transformation of `eval` call becomes independent of the subsequent analysis of the transformed programs.

10.1.2 Related Work

Static analysis of JavaScript has been the focus of much work recently, and the `eval` function is widely recognized as being a challenging language construct.

Thiemann has suggested a type system for detecting suspicious type conversions [89], Anderson et al. have proposed a type inference algorithm for tracking object properties [3], Jang and Choe have presented a points-to analysis for a subset of JavaScript [49], and Logozzo and Venter have introduced an analysis technique that enables type specialization optimizations [63]. All these analyses are defined on subsets of JavaScript that do not include `eval`. The end result is that these analyses currently do not work for many real JavaScript programs.

Guarnieri and Livshits mitigate the problem in the Gatekeeper project by providing a runtime checker that determines if a given JavaScript program falls into the safe subset [33]. Another approach, which is used in the Actarus security analysis tool by Guarnieri et al., is to simply ignore the effects of `eval` [34], which makes analysis results unsound in the presence of `eval` calls.

Dynamically constructed code also presents unique challenges in security analyses that are performed on-the-fly whenever untrusted third-party code is loaded dynamically. Staged or incremental analysis [16, 32] handles the issue by generating security policies that are checked when code is loaded and added to the program using `eval`. In contrast, we can disregard lazy code loading as discussed above, and our approach aims to eliminate `eval` calls by purely static dataflow analysis without runtime checks.

Some uses of `eval` follow common patterns that can be recognized and handled without needing a full analysis. The control flow analysis by Guha et al. recognizes loading of code [35], and our previous work uses similar techniques to rewrite uses of `eval` that simulate simple higher-order functions [50]. In the present work we aim to expand the scope of static analysis for JavaScript in general by transforming `eval` calls into other language constructs that can be handled by existing static analyzers.

We use TAJS [52, 51, 50] to drive the transformation of `eval` calls, but our approach is not inherently tied to TAJS. The general aim of TAJS is to detect likely programming errors related to mismatches of types and dataflow in JavaScript programs, for example to detect suspicious type coercions or function calls where the call expression may not evaluate to a function object. In brief, TAJS performs interprocedural flow-sensitive dataflow analysis with a complex abstract domain that soundly and in great detail models how objects, primitive values, expressions, and statements work in JavaScript according to the ECMAScript standard. Here, we do not use the results produced by TAJS when it analyzes a program; instead we exploit TAJS as a dataflow analysis infrastructure for exposing calls and arguments to `eval`. In previous work [50] we pointed at dynamically generated code as an important next step for static analysis of JavaScript web applications – we here take that step.

The ability to construct code from text at runtime is not limited to JavaScript. Most dynamic scripting languages include an `eval` construct. Furr et al. have presented an intermediate language to ease the task of making static analysis for Ruby [29]. Calls to `eval` are removed using dynamic profiling of the program during the transformation of Ruby programs into this intermediate form [29]. As for the comparison with staged or incremental analysis discussed above, the key difference with our work is that we aim for a sound and purely static approach. Interestingly, the experiments by Furr et al. suggest that `eval` is more commonly used for sophisticated metaprogramming in Ruby programs than Richards et al. have observed in JavaScript programs.

Other programming languages have more disciplined variants of `eval` than the one in JavaScript. As a case in point, `eval` in Scheme [86] works with S-expressions rather than text strings, which makes it easier to reason about the structure of the code being evaluated. Moreover, the code runs in an immutable environment, so it is safe to ignore `eval` calls in static analysis for Scheme, unlike JavaScript.

As mentioned above, our techniques can be viewed as a refactoring that transforms a program to a behaviorally equivalent one without dynamic code evaluation. Similar to the work we present here, Feldthaus et al. use static analysis as a foundation for describing and implementing refactorings of JavaScript programs [26]. One important difference is that we here perform the refactoring during the analysis, not after the analysis fixpoint is reached.

Knowledge about the contents of the strings that are passed to `eval` is obviously essential to be able to transform the `eval` calls to other code. As we show in the following sections, we have chosen a pragmatic approach that aims to cover the patterns that appear to be the most common in practice. This allows us to handle typical calls by focusing on relatively simple patterns of string concatenations. In principle, it would be possible to integrate more advanced string analysis algorithms, as introduced by Christensen et al. [15], but our study of how `eval` is used in practice suggests that our present approach

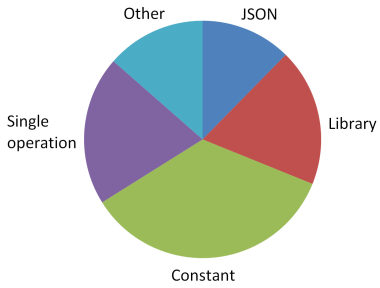


Figure 10.2: Classification of 17,665 `eval` call sites from Alexa top 10,000 web sites.

is adequate in most cases.

10.2 Eval in Practice

To guide the development and to be able to evaluate the quality of our code transformation system, we need a collection of representative example programs that use `eval` and show how it is used in practice. A useful starting point is the study by Richards et al. [78], which is based on execution traces of thousands of the most popular web sites according to Alexa[6]. Their study shows that more than half of the web sites use `eval`, which suggests that there are plenty of examples to choose from. However, we disregard dynamic code loading for the reason mentioned in Section 10.1.1, and JSON parsing can be treated separately with known techniques, which we describe in Section 10.5.2, so these uses of `eval` are less interesting to us. The Richards et al. study does not directly show how many of the web sites use `eval` for purposes other than dynamic code loading and JSON data parsing. Of the remaining uses of `eval`, calls where the argument is a constant string in the source code can also be considered as relatively easy cases for the transformation (we return to this category in Section 10.4).

To investigate this further, we examine the Alexa top 10,000 web sites. We find using the tools made available by Richards et al. that 6,465 of them use `eval`. Filtering out those that use `eval` for purposes other than dynamic code loading and JSON parsing gives us 3,378 URLs. If we further remove those where all calls to `eval` have constant arguments, only 2,589 URLs remain. This alone gives an interesting picture of the typical uses of `eval` that is not emphasized by Richards et al. [78]: Although `eval` is pervasive, we can expect that relatively few web sites (around 25%) use `eval` in ways that are truly challenging to reason about with static analysis.

A second observation is that the results of measuring the `eval` usage patterns are more useful to us if we count numbers of static call sites rather than numbers of runtime calls to `eval` as in the Richards et al. study. Many calls at runtime typically originate from the same call sites in the code, and for the purpose of developing techniques to transform source code to eliminate typical `eval` calls, we obtain more relevant information about the usage patterns by considering the static call site information. Of a total of 17,665 `eval` call sites,

[6] `http://www.alexa.com/topsites`

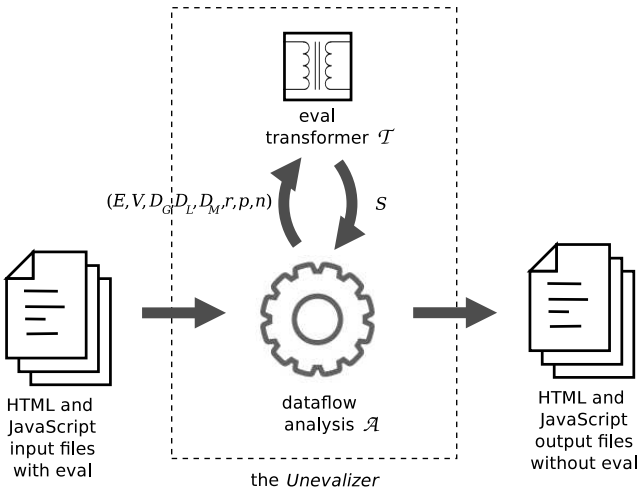


Figure 10.3: Structure of the *Unevalizer*.

we find that 3,339 are used for loading library code, 6,228 have arguments that are constant strings (see Section 10.4), and 2,202 are used for parsing JSON data (see Section 10.5.2). Of the remaining call sites, 3,624 evaluate code strings that are single operations, such as property read/write operations, `typeof` type test expressions, or simple function/method calls. A few call sites, 141, fall into more than one of these categories. The distribution is shown in Figure 10.2. This suggests that a transformation technique that can handle constants, JSON, and single operations will cover a majority of the `eval` calls that programmers write.

10.3 The Unevalizer Framework

Figure 10.3 shows the structure of the *Unevalizer*. As input, it takes a JavaScript web application containing HTML and JavaScript files. It then transforms the application driven by a whole-program dataflow analysis and, if successful, outputs a semantically equivalent application that does not contain calls to `eval`.

The dataflow analysis $\mathcal{A}$ will abstractly trace all possible execution paths through the program and keep track of what data flows into what variables and functions. This process is based on the classical monotone framework [56] that maintains abstract states for all program points and abstract values for all expressions. Specifically, it models function objects using object labels where $\ell_{\texttt{eval}}$ is an object label describing the `eval` function that is defined in the ECMAScript core library. Our prototype implementation uses TAJS for the dataflow analysis. Whenever new dataflow is detected during the analysis at a function call site `F(E)`, where F and E are expressions, we look for calls to `eval`: If the abstract value provided by the analysis for F includes $\ell_{\texttt{eval}}$ then the transformation component $\mathcal{T}$ is triggered. Method calls are treated similarly as function calls, and we omit them here to simplify the presentation. We also

ignore indirect calls via built-in native functions such as `call` and `apply`, which are fully supported by our analysis but rarely used in combination with `eval`.

The transformation component $\mathcal{T}$ is passed an 8-tuple $(E, V, D_G, D_L, D_M, r, p, n)$ with information from the analysis:

- E is the syntactic argument expression as it appears in the program code at the function call site.
- V is the abstract value of the argument expression E. This abstract value soundly approximates the code string to be evaluated.
- D_G and D_L are the sets of names of variable and function declarations in the global and local scope, respectively. This takes into account nesting of functions and properties of the global object. D_M is the set of names of built-in properties of the global object that may have been modified by the application code. We settle for sound approximations of these sets since JavaScript does not have ordinary lexical scope (due to `with` statements and dynamically constructed properties of the global object that are always in scope).
- r is a boolean flag that indicates whether the call appears syntactically as an expression where its return value is used (as in `x=eval(y)`) or as a statement on its own.
- p is a boolean flag that signals whether the `eval` call is direct or aliased, which controls its execution scope as mentioned in Section 10.1.
- n is a number that indicates the `eval` nesting depth, which is 0 for an `eval` call that occurs in the original source program, 1 for a call that appears in code generated by an `eval` call at nesting depth 0, etc.

This turns out to be sufficient information to perform the transformation in many common cases. Note that for a given call site we can statically determine E, r, and p from the syntax of the call and its context, whereas V, D_G, D_L, D_M, and n may vary during the analysis. We assume that the underlying dataflow analysis models possible string values of expressions using a finite-height lattice Str. We discuss specific choices of this lattice in Sections 10.4 and 10.5. On top of this, we give special treatment to argument expressions that are built from concatenations using the `+` operator, which is common in practice. As an example, for the call

```
1  eval("v"+i+"="+x)
```

the argument expression E is `"v"+i+"="+x` and its abstract value V is $v_1 \oplus v_2 \oplus v_3 \oplus v_4$ where each $v_1, \ldots, v_4 \in \mathsf{Str}$ are abstract values of the four constituents and $\oplus$ represents concatenation. Note that we do not require the underlying dataflow analysis to reason precisely about string concatenations, and the $\oplus$ operator is only used to model concatenations that appear literally in the argument expression E.

In response, $\mathcal{T}$ gives either

- a string S containing JavaScript code that is equivalent to the function call `F(E)` relative to the given context, or
- the special value $\lightning$ in case it is unable to transform the given `eval` call.

There will inevitably be situations where $\lightning$ is returned, for example if the value of E partly originates from the user via an HTML text field, as discussed in Section 10.1.

If T returns successfully, the *Unevalizer* will incorporate S into the code base at the point of the function call and proceed with the analysis. In doing this, we must consider the possibility that $\ell_{\texttt{eval}}$ may not be the only value of F, in which case the analysis must process all the possible functions and join their respective abstract return states. Additionally, we must take into account the fact that E may evaluate to non-string values. Such arguments to `eval` are simply returned directly without string coercion according to the ECMAScript specification. Moreover, we must retain the original call $F(E)$ in the code since more dataflow may appear later in the analysis, which triggers new invocations of $\mathcal{T}$. Consider the following example:

```
1  if (...)
2    x = "f";
3  else
4    x = "g";
5  ...
6  eval(x + "()");
```

The first time dataflow arrives at the `eval` call site, it is possible that $\mathcal{A}$ has the information that the value of `x` is the string `"f"`, which could result in S becoming the code `f()`. However, $\mathcal{A}$ will later realize that `"g"` is also a possible value of `x`, and this may cause a different output from $\mathcal{T}$ replacing the old value of S.

As common in dataflow analysis using the monotone framework, the *Unevalizer* operates as a fixpoint computation that starts with the empty abstract states and empty abstract values everywhere and then applies monotone transfer functions iteratively until the least fixpoint is reached [56]. When $\mathcal{A}$ encounters a call to `eval`, that gets replaced by the code S, which $\mathcal{A}$ subsequently models as an abstract transformer $\widehat{S}$ relative to the abstract domain in use. This informally explains how we avoid the apparent chicken-and-egg problem we mentioned in Section 10.1: At each call site where $\ell_{\texttt{eval}}$ occurs, the corresponding values V, D_G, D_L, and D_M grow monotonically during the process. This, however, requires the transformation component to be monotone in the following sense:

Property 1 (Monotonicity)**.** *Let* $C = (E, V, D_G, D_L, D_M, r, p, n)$ *and* $C' = (E, V', D'_G, D'_L, D_M, r, p, n)$ *be inputs to* $\mathcal{T}$ *such that* $V \sqsubseteq_{\mathsf{Value}} V'$, $D_G \subseteq D'_G$, $D_L \subseteq D'_L$, $D_M \subseteq D'_M$ *where* $\sqsubseteq_{\mathsf{Value}}$ *is the partial order of abstract values in the dataflow analysis, and let* S *and* S' *denote the outputs from* $\mathcal{T}$*, that is,* $S = \mathcal{T}(C)$ *and* $S' = \mathcal{T}(C')$*. Let* $\widehat{S}$ *and* $\widehat{S}'$ *denote the corresponding abstract transformers with respect to the abstract domain used by* $\mathcal{A}$*. The transformation component* $\mathcal{T}$ *is* monotone *in the senses that* $\widehat{S} \sqsubseteq_{\mathsf{Trans}} \widehat{S}'$ *for any such two inputs* C *and* C' *where* $\sqsubseteq_{\mathsf{Trans}}$ *is the partial order of the abstract transformers.*

As the *Unevalizer* replaces calls to `eval` with other code that is analyzed subsequently, we must be careful with generated code that itself calls `eval`, although that is not common in practice. An example from `bild.de`:

```
1  eval("try { lFrame = eval(lf[i]) }catch(e){};");}
```

The `eval` nesting depth n gives us an easy way to ensure that the *Unevalizer* always terminates:

Property 2 (Convergence). *If $n > k$ for some bound k then $\mathcal{T}$ returns $\lightning$.*

The bound $k = 1$ suffices for all examples we have encountered.

We can now establish the meaning of correctness for the *Unevalizer* and the requirements to $\mathcal{A}$ and $\mathcal{T}$:

Property 3 (Correctness). *Assuming that*

- *the underlying dataflow analysis $\mathcal{A}$ is sound,*
- *for any input $(E, V, D_G, D_L, D_M, r, p, n)$, $\mathcal{T}$ outputs either $\lightning$ or a program fragment S that has the same external behavior as the call `eval(E)` in the context given by V, D_G, D_L, D_M, r, and p, and*
- *$\mathcal{T}$ satisfies Properties 1 and 2,*

the Unevalizer is guaranteed to output a program that has the same external behavior as the input, or report that it is unable to transform the input.

Upon completion, the *Unevalizer* outputs JavaScript and HTML files where all calls to `eval` have been eliminated. This allows the output to be be further analyzed by other analyses that do not work on programs that contain `eval`. In the following two sections we describe our instantiations of the framework.

10.4 Eliminating Calls to Eval with Constant Arguments

We start by introducing techniques needed to remove calls to `eval` where the argument E is a constant string. Surprisingly many programs actually call `eval` with constant string arguments, as observed in Section 10.2. More importantly, this transformation is used as a stepping stone for Section 10.5 where we consider more general `eval` calls.

The task might appear trivial, but there are several issues to consider to ensure that the transformation is correct. A naive approach of simply "dropping the quotes" may yield a program with a different behavior. Consider the following hypothetical rewrite rule:

`eval("var x;")` $\rightsquigarrow$ `var x;`

This rule might appear correct at a first glance, but consider the `eval` call below and the resulting program after applying the transformation:

```
1 var x = 2;
2 function f() {
3   var y = x;
4   eval("var x;"
        );
5   return y;
6 }
7 f();
```

$\rightarrow$

```
1 var x = 2;
2 function f() {
3   var y = x;
4   var x;
5   return y;
6 }
7 f();
```

These two programs are not equivalent: the one on the right yields `undefined` rather than `2` since the global variable `x` is shadowed by the local with the same name.

In general, the following five issues must be considered when transforming `eval` calls with constant strings.

Statements When the `eval` call occurs as an expression and E consists of statements rather than a single expression, the code must be reorganized using temporary variables to ensure a correct order of evaluation. For example,

```
x = a() * eval("b(); c();") * d();
```

can be translated into the following code:

```
var t1 = a();
b();
var t2 = c();
x = t1 * t2 * d();
```

This raises a subtle issue about generating fresh names, here `t1` and `t2`. We pick names that are not in $D_G \cup D_L$, or return $\lightning$ in case that set contains all possible identifier names.

Declarations Function and variable declarations in E can potentially clash with identifiers already in scope, as shown by the example in the beginning of this section. Since D_G and D_L are available during the analysis, we simply let $\mathcal{T}$ return $\lightning$ if any new variable declarations in E are already in $D_G \cup D_L$.

Syntactic Validity If the string passed to `eval` at runtime is not a syntactically valid program, a `SyntaxError` exception is thrown. This is easy for $\mathcal{T}$ to check when the string is a constant, simply by running a JavaScript parser. If the string is invalid, $\mathcal{T}$ returns $S =$ `throw new SyntaxError()`. The name `SyntaxError` may, however, be shadowed, so if `SyntaxError` $\in D_L \cup D_M$, we instead let $\mathcal{T}$ return $\lightning$. Although this is unlikely a problem in practice, it is necessary for soundness.

Return Value The return value of `eval` is defined to be the value of the last so-called value yielding statement executed in the input string. Most statements have a value, however a few such as the empty block and `var` statements do not. This means that the return value of an `eval` call cannot always be statically determined, even if the entire input string is a known constant. Consider for example this call:

```
eval("2;if (b) 3;")
```

Its return value is either `2` or `3` depending on the value of the `b` variable. Rather than trying to devise complex transformation rules to handle such cases, we choose a simple alternative that seems to suffice in practice: If the return value is not used, which $\mathcal{T}$ knows from the r flag, then there is no issue. Otherwise, we let $\mathcal{T}$ return $\lightning$ if it is ambiguous which statement will yield the return value. The string has already been parsed at this point, as discussed above, so checking for this kind of ambiguity is straightforward.

Scope Another peculiar corner case in the ECMAScript standard is that the execution scope of dynamically evaluated code depends on whether `eval` is called directly or through an alias, which was the reason for introducing the p flag in Section 10.3. The following example uses an alias for `eval` to access a variable `x` in the global scope, even if the variable name `x` is shadowed by a local declaration:

```
1 var geval = eval;
2 geval("x = 5");
```

When the p flag is set to global scope execution, $\mathcal{T}$ needs to transform the code to ensure the proper binding of identifiers. At first, one may try to exploit the fact that the global object is a synonym for `window`, however the `window` variable may itself be overwritten or shadowed by local declarations. A more robust way to get a reference to the global object is to evaluate the expression `function (){return this;})()`, which we abbreviate as *global*. This is perhaps not pretty but it satisfies our requirement of being analyzable with, for example, TAJS. The call `geval("x = 5")` in the example above is then transformed into *global*`.x = 5`. Declarations in the global scope can be transformed similarly, for example `geval("function f(){...}")` becomes *global*`.f = function(){...}`.

One additional issue remains. Reading a nonexistent variable in JavaScript will throw a `ReferenceError`, but reading an absent property just yields the value `undefined`. If we change an identifier read operation naively into a property read operation, for example from `geval("x")` to *global*`.x`, the behavior changes if the identifier is undeclared. Instead we transform it into a conditional expression:

```
1 "x" in global ? global.x : throw new ReferenceError()
```

and check whether `ReferenceError` has been overwritten, as for `SyntaxError` earlier in the section.

10.5 More Precise Analysis of the Arguments to Eval

Eliminating calls to `eval` with constant arguments as done in Section 10.4 handles the tip of the iceberg. We now suggest four pragmatic ways of building on top of the transformation described in the previous sections by more deeply exploiting the connection between the transformation component and the dataflow analysis.

10.5.1 Exploiting Constant Propagation

We obtain the first improvement using constant propagation, which the TAJS dataflow analysis already performs. Technically, the Str lattice mentioned in Section 10.3 contains an unordered set of all possible string constants and a top element $\top$ representing non-constant strings, and all transfer functions in TAJS are designed to perform constant folding.

The following example extracted from the web site `qq.com` demonstrates an `eval` call where simple constant propagation is enough to enable transformation:

```
1 var json = "<large constant string>";
2 ...
3 eval("area="+json);
```

Consider also the following example from the Chrome Experiments program `canvas-sketch`[7] that uses `eval` to emulate higher-order functions:

[7] `http://www.chromeexperiments.com/detail/canvas-sketch/`

```
1 if (vez.func instanceof Function) vez.func(texto);
2 else eval(vez.func + "(texto)");
```

It turns out that interprocedural constant propagation for this program is able to infer that `vez.func` is always a constant string. To handle an even larger class of `eval` calls, in Section 10.5.4 we present a way to boost the effect of constant propagation using code specialization.

10.5.2 Tracking JSON Strings

JSON is a standardized format for data exchange that is derived from the JavaScript syntax for objects, arrays, and primitive values [19]. It is designed such that JSON data can be parsed using `eval`, and many `eval` calls are used for this purpose as discussed in Section 10.2. Modern browsers have the function `JSON.parse` for parsing the JSON subset of JavaScript in a more safe and efficient manner. Many programs check whether the `JSON` object exists and, if not, fall back to calling `eval` for parsing JSON data.

The following pattern occurs in many web sites:

```
1 x = eval("(" + v + ")");
```

The wrapping forces `v` to be evaluated as an expression. If `v` contains JSON data, this `eval` call can be translated as follows:

```
1 x = JSON.parse(v);
```

The benefit of this transformation is that `JSON.parse`, unlike `eval`, never has side-effects other than creating an object structure, so it can easily be modeled soundly in a static analysis.

We use the technique introduced in our earlier work [50] to find out which values contain JSON data: The Str lattice is augmented with a special abstract value JSONString that represents all strings that are valid JSON data. The transformation suggested above can then be applied whenever the abstract value V of E is, e.g., `"("` $\oplus$ JSONString $\oplus$ `")"`.

Now, the problem is to detect when JSON data is created. This is easy for constant strings and for the function `JSON.stringify` that explicitly constructs JSON data, however the most common source of JSON data is Ajax communication with the server. Since we cannot know what data the server produces by only analyzing the client-side of the web application, we choose to rely on user annotations in the JavaScript code to specify sources of JSON data, typically in Ajax response callbacks.

JSON data obtained using Ajax is in rare situations combined with other string values before being passed to `eval`. We leave it to future work to incorporate more elaborate string analysis [15] for reasoning about such cases.

10.5.3 Handling Other Non-Constant Strings

It is evident from Figure 10.2 that we need to handle other cases than constants and JSON strings. A common pattern is `eval("foo."+x)` that accesses a property of an object. This can be transformed into `foo[x]`, but only if we can be certain that `x` evaluates to specific classes of values, such as numbers or strings that are valid identifier names. The transformation would be unsound if `x` has a value such as `"f*2"`. This example suggests that we refine the Str lattice further: we

introduce a new abstract value IdString representing all strings that are valid JavaScript identifiers. TAJS handles number values in a similar way as strings, so we here focus on the string values.

Related patterns such as `eval("foo_"+x)` and `eval(x+"_foo")`, which also appear in widely used web applications, can be handled similarly. However, in the case of `eval("foo_"+x)` we can loosen the requirement on `x`. It suffices to know that `x` is a string that consists of characters that are valid in identifiers, excluding the initial character. We therefore extend Str with yet another abstract value IdPartsString representing such strings. As an example, the string `"42"` belongs to IdPartsString but not to IdString.

With these extensions, the *Unevalizer* can handle cases such as this one from `canvas-cycle` where $\mathcal{A}$ infers the abstract value IdString for the variable `key`:

```
1 eval('window.' + key + ' = {};');
```

In the following example from the web site `zedo.com` the abstract value of `v0[i]` is IdPartsString:

```
1 for(var i=0;i<v0.length;i++){
2  if(eval("typeof(zflag_"+v0[i]+")!='undefined'")){ ...
```

When transforming calls such as `eval("foo_"+x)` that access identifiers with computed names we run into the problem described in Section 10.1.1 that JavaScript does not provide a general mechanism for accessing the current scope object, so we restrict ourselves to the cases where we are certain that the identifiers are not bound locally: if D_L contains names that in this case start with `"foo_"` then $\mathcal{T}$ returns $\lightning$.

10.5.4 Specialization and Context Sensitivity

By selectively exploiting context sensitivity of the dataflow analysis the *Unevalizer* can also handle many `eval` calls where the strings are not constant but can be traced to a finite number of constant sources. Consider the following representative example from the web site `fiverr.com`:

```
1  get_cookie = function (name) {
2    var ca = document.cookie.split(';');
3    for (var i = 0, l = ca.length; i < l; i++) {
4      if (eval("ca[i].match(/\\b" + name + "=/)"))
5        return decodeURIComponent(ca[i].split('=')[1]);
6    }
7    return '';
8  }
9  get_cookie('clicky_olark')
10 get_cookie('no_tracky')
11 get_cookie('_jsuid')
```

When the analysis enters `get_cookie` from the first call site, the `name` parameter will be bound to the constant string `"clicky_olark"`. Constant propagation to the `eval` call will then enable transformation as in Section 10.4. When the analysis later encounters the second call to `get_cookie`, the `name` parameter would with a context insensitive analysis obtain the abstract value IdString, which would flow to the `eval` call and cause $\mathcal{T}$ to fail with $\lightning$. Instead, when `name` first flows to the `eval` call we mark that `get_cookie` shall be analyzed context sensitively with respect to the `name` parameter. This will ensure that

the second and the third call to `get_cookie` with different arguments will be analyzed separately. As a result, the analysis will know that the only possible values of `name` at the `eval` call site are `"clicky_olark"`, `"no_tracky"`, and `"_jsuid"`. This can be used to specialize the argument to `eval` and transform the `eval` call into the following expression:

```
1 name==="clicky_olark" ? ca[i].match(/\\bclicky_olark=/)
2 : name==="no_tracky" ? ca[i].match(/\\bno_tracky=/)
3 : ca[i].match(/\\b_jsuid=/)
```

This mechanism can in principle be taken a step further to handle situations where the `eval` call appears nested inside more function calls, similar to k-CFA or the use of call strings in interprocedural analysis [70], however, one level of selective context sensitivity seems to suffice in our setting.

10.6 Evaluation

We have implemented the `eval` transformer $\mathcal{T}$ and use TAJS as the driving dataflow analysis, $\mathcal{A}$. The two are cleanly separated by an interface similar to the 8-tuple described in Section 10.3. Any program implementing this interface can in principle use the transformation component.

In this section we describe our experiences running the prototype on a benchmark collection. We will try to answer the following research questions about the *Unevalizer*.

Q1: Is the *Unevalizer* able to transform common usage patterns of `eval` calls?

Q2: To what extent are the individual techniques presented in Sections 10.4 and 10.5 useful in practice?

Q3: For call sites where the *Unevalizer* fails to find a valid transformation, can we suggest improvements that are likely to handle more cases?

10.6.1 Benchmarks

Our main source of benchmarks is the Alexa list[6] that we also used in Section 10.2. We focus on the most challenging cases of `eval`, which are the call sites that fall into the categories "other" or "single operation" described in Section 10.2. We exclude all web sites that do not have any instances of `eval` in these categories. Library loading is outside the scope of this work as discussed in Section 10.1.1, and the technique we use for JSON data in Section 10.5.2 has to some extent been covered before [50]. Applying these criteria on the Alexa top 500 list gives us 19 web sites.

Analyzing JavaScript web applications involves many other challenges than `eval`. Although TAJS is able to analyze many real applications [50], the 19 applications collected from the Alexa list are still beyond the current capabilities of TAJS because they are considerably larger than what we have run TAJS on previously. However, since the purpose of the present evaluation is not to test the quality of TAJS but how the *Unevalizer* performs, we choose to manually extract the parts of the web applications that involve calls to `eval` including the relevant dataflow. This exposes 25 interesting program slices, each containing one or more calls to `eval`.

Our previous experiments with TAJS considered programs from Chrome Experiments[5], which generally have more manageable sizes than the Alexa top

Site	Call Sites	ConstProp	Identifier	Specialization	Pass
`berts-breakdown`	1	-	-	-	×
`canvas-cycle`	1	-	-	1	✓
`canvas-sketch`	1	1	-	-	✓
`bild.de` (1)	1	-	1	-	✓
`bild.de` (2)	1	-	-	-	×
`conduit.com`	1	-	-	1	✓
`dailymotion.co.uk`	1	1	-	-	✓
`fiverr.com`	1	-	-	1	✓
`huffpost.com`	1	-	-	-	×
`imdb.com`	2	2	-	-	✓
`indiatimes.com`	2	2	-	-	✓
`myspace.com`	1	-	-	1	✓
`onet.pl` (1)	1	-	-	-	×
`onet.pl` (2)	1	-	-	-	×
`pconline.com.cn` (1)	1	-	-	1	✓
`pconline.com.cn` (2)	1	-	-	-	×
`rakuten.co.jp`	1	1	-	-	✓
`scribd.com`	2	-	-	2	✓
`sohu.com`	2	-	-	2	✓
`telegraph.co.uk` (1)	1	-	-	-	×
`telegraph.co.uk` (2)	2	-	2	-	✓
`washingtonpost.com`	1	-	-	-	×
`wp.pl`	1	1	-	-	✓
`xing.com`	3	-	-	-	×
`xunlei.com`	6	6	-	-	✓
`zedo.com` (1)	3	-	3	-	✓
`zedo.com` (2)	3	-	3	-	✓
`zedo.com` (3)	1	1	-	-	✓
Total	44	15	9	9	

Table 10.1: Experimental results. The first three programs are the ones from Chrome Experiments; the remaining ones are the sliced programs from the Alexa list. The columns "Call Sites" shows the number of `eval` calls, the next three columns show which techniques the *Unevalizer* uses to transform the calls, and the "Pass" column shows which programs are transformed successfully.

500 web sites. We have found 3 programs in Chrome Experiments that use `eval` in ways that satisfy the criteria mentioned above, and we include those programs unaltered without slicing.

The resulting 28 programs are listed in Table 10.1. For each of the sliced web sites, we list each program slice separately. The benchmark collection can be downloaded from `http://www.brics.dk/TAJS/unevalizer-benchmarks`.

10.6.2 Experiments

In this section we describe the experiments used to answer research questions Q1 and Q2. The last question, Q3, is discussed in Section 10.6.3.

Q1 is adressed by the column "Pass" in Table 10.1. The symbol ✓ indicates that the *Unevalizer* is able to successfully transform all `eval` call sites in the program, and × means that $\mathcal{T}$ returns $\lightning$ at some point during the fixpoint

computation. We see that the *Unevalizer* is able to handle 19 out of 28 cases, corresponding to 33 out of 44 `eval` call sites.

We address Q2 with the three columns "ConstProp", "Identifier" and "Specialization" in Table 10.1. The numbers in those columns show how many call sites are handled by each of the three techniques presented in Sections 10.5.1, 10.5.3, and 10.5.4, respectively. Note that the specialization technique builds on top of constant propagation, but the numbers for "ConstProp" only include the cases that do not also require specialization.

We see that out of 44 call sites, constant propagation (Section 10.5.1) alone is enough to transform 15 `eval` call sites. Using identifier detection (Section 10.5.3) we eliminate 9 more call sites, and if we also add specialization (Section 10.5.4) 9 additional call sites are successfully transformed. These numbers suggest that all the techniques we have presented are useful in practice.

Example

An example of a successful transformation is `sohu.com`, which uses `eval` to create a form of dynamic dispatch based on property names in objects. The two `eval` calls appear in the same function `_SoAD_exec`:

```
1  function _SoAD_exec(o) {
2   if (eval("typeof(" + o.t + "_main)") == "function")
3     eval(o.t + "_main(o)");
4  }
```

The dataflow analysis determines from the call sites to the function `_SoAD_exec` that `o.t` has the abstract value IdString. Using the techniques in Sections 10.4 and 10.5.3, the sub-expression `o.t+"_main"` can be rewritten into a property read operation on the global object. To guard against potential clashes with identifiers in the local scope, the *Unevalizer* checks that no names in D_L have the suffix `"_main"`. The second `eval` call site is transformed in a similar manner. The resulting function looks as follows:

```
1  function _SoAD_exec(o) {
2    if (typeof(
3          (o.t + "_main") in global ?
4              global[o.t + "_main"] :
5              throw new ReferenceError())
6         == "function")
7        ((o.t + "_main") in global ?
8              global[o.t + "_main"] :
9              throw new ReferenceError())(o);
10 }
```

In this code *global* refers to the expression that returns the global object, as defined in Section 10.4. The conditional expressions ensure that a `ReferenceError` is thrown if the property is absent in the global object.

Threats to Validity

The fact that the *Unevalizer* successfully eliminates many nontrivial `eval` calls in some manually extracted program slices and a few medium size complete web applications obviously does not imply that all problems related to `eval` are now solved. Our manual slicing may be erroneous although we have strived to

preserve all dataflow that is relevant for the `eval` call sites. Ideally, we would of course like to test our approach on a larger number of web applications and on the complete application code without slicing, but, as mentioned in Section 10.6.1, that requires a more scalable dataflow analysis than the current version of TAJS. With today's state-of-the analysis techniques for JavaScript, we see no better way of evaluating the *Unevalizer* than using the slicing approach. Also, the programs included in the evaluation are all from real web sites and have been selected in a systematic and non-biased manner, following the criteria described in Section 10.6.1 that have exposed the most interesting cases of `eval`. We also point out that the *Unevalizer* can leverage from future improvements of TAJS or other dataflow analyses for JavaScript.

A second concern could be that the web sites from the Alexa list, which was also the foundation for Richards et al. [78], and the Chrome Experiments may not be representative for JavaScript web applications in general, however we believe the programs included in the evaluation give a good indication of how `eval` is being used in practice.

10.6.3 Directions for Future Improvements

To answer Q3 we examine the cases where the *Unevalizer* fails to transform an `eval` call site. Overall we observe two reasons for failure: insufficient precision of the dataflow analysis on loop control structures (this accounts for 6 of the 11 failing `eval` call sites), and `eval` call sites where the argument is built from string concatenations that do not appear syntactically inside at the function call (4 cases).

Loops seem to cause a loss of precision that often hinders transformation. The following example from the web site `bild.de` demonstrates such a case:

```
1 for (var libName in $iTXT.js.loader) {
2   currentLibName = libName;
3   eval(libName + '_Load()');
4 }
```

The loop iterates over all the properties of an object, which is defined by a constant object literal elsewhere in the code. The property names do not match IdStrings, however, so the abstract value of `libName` becomes $\top$, which is insufficient to transform the `eval` call site. Applying loop unrolling in $\mathcal{A}$ to this example would enable better constant propagation, which could in turn enable transformation of the call site.

Recall from Section 10.3 that we give special treatment to string concatenations that appear syntactically in the `eval` argument expressions. This works well for the majority of our benchmarks, however the following example from `pconline.com.cn` shows a situation where it is inadequate:

```
1 function showIvyViaJs(locationId) {
2   ...
3   var _fconv = "ivymap[\'"+locationId+"\']";
4   try {
5     _f = eval(_fconv);
6     ...
7   } catch(e) {}
8 }
```

The string given to `eval` is created from concatenations, but not at the call site, and the abstract domain Str for string values in TAJS is not detailed enough

to model the possible values of `_fconv` with sufficient precision. The abstract value V then becomes $\top$, which causes the *Unevalizer* to give up. One way to improve this would be to extend the constant propagation in $\mathcal{A}$ to propagate entire expressions. In the example, this could propagate the expression `"ivymap [\'"+locationId+"\']"` directly into the `eval` call, and then $\mathcal{T}$ would be able to handle it. Propagating expressions in a sound way is not trivial, however, as the order of evaluation must be preserved for certain operations.

Notice that both of the improvements suggested in this section could be implemented entirely inside $\mathcal{A}$ without modifying $\mathcal{T}$ or the general *Unevalizer* framework.

10.7 Conclusion

The `eval` function is in practice not as evil as some men claim. By incorporating an `eval` elimination refactoring into a dataflow analysis, we have demonstrated that it is often possible to eliminate calls to `eval` in a sound and automated manner and thereby enable static analysis of JavaScript programs that use `eval` in nontrivial ways. Although we base our proof-of-concept implementation, the *Unevalizer*, on the TAJS dataflow analysis infrastructure, our approach is not intimately tied to the inner workings of TAJS: any dataflow analysis that can safely provide the necessary information to the transformation component could be used. It is also possible to apply other analyses to the resulting program code, including many of those mentioned in Section 10.1.2.

Our experimental results suggest that the approach succeeds in eliminating typical uses of `eval`, but also that further improvements are likely possible within the framework. Our future work will focus on the challenges related to `eval` calls that appear in loops and on extending constant propagation to handle entire expressions, as suggested in Section 10.6.3. Furthermore, now that many more JavaScript web applications are within range of static analysis, it becomes possible to explore new opportunities for improving other aspects of static analysis techniques for JavaScript.

Acknowledgments

This work was supported by Google, IBM, and The Danish Research Council for Technology and Production.

Bibliography

[1] Adobe. *JSEclipse*. `http://labs.adobe.com/technologies/jseclipse/`.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., 1986. ISBN: 0-201-10088-6.

[3] Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. "Towards Type Inference for JavaScript". In: *Proc. 19th European Conference on Object-Oriented Programming, ECOOP '05*. Vol. 3586. LNCS. Springer-Verlag, 2005.

[4] Shay Artzi, Julian Dolby, Simon Holm Jensen, Anders Møller, and Frank Tip. "A Framework for Automated Testing of JavaScript Web Applications". In: *Proc. 33rd International Conference on Software Engineering, ICSE '11*. 2011.

[5] Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Danny Dig, Amit M. Paradkar, and Michael D. Ernst. "Finding bugs in dynamic web applications". In: *Proc. International Symposium on Software Testing and Analysis, ISSTA '08*. ACM, 2008.

[6] Darren C. Atkinson and William G. Griswold. "Implementation techniques for efficient data-flow analysis of large programs". In: *Proc. International Conference on Software Maintenance, ICSM '01*. 2001, pp. 52–61.

[7] Gogul Balakrishnan and Thomas W. Reps. "Recency-Abstraction for Heap-Allocated Storage". In: *Proc. 13th International Static Analysis Symposium, SAS '06*. Vol. 4134. LNCS. Springer-Verlag, 2006.

[8] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. "A Static Analyzer for Large Safety-Critical Software". In: *Proc. of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation PLDI'03*. 2003, pp. 196–207.

[9] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. "Design and Implementation of a Special-Purpose Static Program Analyzer for Safety-Critical Real-Time Embedded Software, invited chapter". In: *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones.*

Ed. by T. Mogensen, D.A. Schmidt, and I.H. Sudborough. LNCS 2566. Springer-Verlag, Oct. 2002, pp. 85–108. ISBN: 3-540-00326-6.

[10] Norris Boyd et al. *Rhino: JavaScript for Java*. `http://www.mozilla.org/rhino/`.

[11] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. "A Static Analyzer for Finding Dynamic Programming Errors". In: *Software: Practice and Experience* 30.7 (2000). John Wiley & Sons, pp. 775–802.

[12] Robert Cartwright and Mike Fagan. "Soft Typing". In: *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '91*. 1991.

[13] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. "Analysis of pointers and structures". In: *Proc. of the ACM SIGPLAN 1990 conference on Programming language design and implementation, PLDI '90*. 1990, pp. 296–310.

[14] Bradley Childs. *JavaScript Development Toolkit (JSDT) Features*. July 2008. URL: `http://live.eclipse.org/node/569`.

[15] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. "Precise Analysis of String Expressions". In: *Proc. 10th International Static Analysis Symposium, SAS '03*. Vol. 2694. LNCS. Springer-Verlag, 2003, pp. 1–18.

[16] Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. "Staged information flow for JavaScript". In: *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '09*. 2009.

[17] Patrick Cousot and Radhia Cousot. "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints". In: *Proc. 4th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '77*. 1977, pp. 238–252.

[18] Douglas Crockford. *JavaScript: The Good Parts*. O'Reilly, 2008. ISBN: 978-0-596-51774-8.

[19] Douglas Crockford. *RFC 4627 - The application/json Media Type for JavaScript Object Notation (JSON)*. URL: `http://tools.ietf.org/html/rfc4627`.

[20] Olivier Danvy and Lasse R. Nielsen. *Refocusing in Reduction Semantics*. rs. BRICS, 2004.

[21] B. A. Davey and H. A. Priestly. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[22] Julian Dolby. *Using Static Analysis for IDE's for Dynamic Languages*. The Eclipse Languages Symposium. 2005.

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

[24] Dawson R. Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. "Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions". In: *4th Symposium on Operating System Design and Implementation, OSDI '00*. USENIX, 2000.

[25] Christian Fecht and Helmut Seidl. "Propagating Differences: An Efficient New Fixpoint Algorithm for Distributive Constraint Systems". In: *Programming Languages and Systems, Proc. 7th European Symposium on Programming, ESOP '98*. Vol. 1381. LNCS. Springer-Verlag, 1998.

[26] Asger Feldthaus, Todd Millstein, Anders Møller, Max Schäfer, and Frank Tip. "Tool-supported Refactoring for JavaScript". In: *Proc. 26th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA '11*. 2011.

[27] Stephen Fink and Julian Dolby. *WALA – The T.J. Watson Libraries for Analysis*. URL: `http://wala.sourceforge.net/`.

[28] Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Stephanie Weirich, and Matthias Felleisen. "Catching Bugs in the Web of Program Invariants". In: *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '96*. 1996, pp. 23–32.

[29] Michael Furr, Jong hoon (David) An, Jeffrey S. Foster, and Michael W. Hicks. "Static type inference for Ruby". In: *Proc. ACM Symposium on Applied Computing, SAC '09*. 2009.

[30] Patrice Godefroid, Nils Klarlund, and Koushik Sen. "DART: Directed automated random testing". In: *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation, PLDI '05*. 2005.

[31] Justin O. Graver and Ralph E. Johnson. "A type system for Smalltalk". In: *Proc. 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '90*. 1990, pp. 136–150.

[32] Salvatore Guarnieri and Benjamin Livshits. "Gulfstream: Staged Static Analysis for Streaming JavaScript Applications". In: *Proc. USENIX Conference on Web Application Development, WebApps '10*. 2010.

[33] Salvatore Guarnieri and V. Benjamin Livshits. "Gatekeeper: Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code". In: *Proc. 18th USENIX Security Symposium, Security '09*. 2009.

[34] Salvatore Guarnieri, Marco Pistoia, Omer Tripp, Julian Dolby, Stephen Teilhet, and Ryan Berg. "Saving the world wide web from vulnerable JavaScript". In: *Proc. 20th International Symposium on Software Testing and Analysis, ISSTA '13*. ACM, 2011.

[35] Arjun Guha, Shriram Krishnamurthi, and Trevor Jim. "Using static analysis for Ajax intrusion detection". In: *Proc. 18th International Conference on World Wide Web, WWW '09*. 2009.

[36] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. "The essence of javascript". In: *Proc. of the 24th European conference on Object-oriented programming, ECOOP 10*. LNCS. Springer-Verlag, 2010, pp. 126–150.

[37] Arjun Guha, Claudiu Saftoiu, and Shriram 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, 2011.

[38] Brian Hackett and Shu yu Guo. "Fast and precise hybrid type inference for JavaScript". In: *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12*. 2012, pp. 239–250.

[39] Phillip Heidegger and Peter Thiemann. "Recency Types for Analyzing Scripting Languages". In: *Proc. 24th European Conference on Object-Oriented Programming, ECOOP '10*. LNCS. Springer-Verlag, 2010.

[40] Phillip Heidegger and Peter Thiemann. "Recency Types for Dynamically-Typed Object-Based Languages". In: *Proc. International Workshops on Foundations of Object-Oriented Languages, FOOL '09*. Jan. 2009.

[41] Michael Hind. "Pointer analysis: haven't we solved this problem yet?" In: *Proc. ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE '01*. 2001, pp. 54–61.

[42] Michael Hind, Michael G. Burke, Paul R. Carini, and Jong-Deok Choi. "Interprocedural pointer alias analysis". In: *ACM Transactions on Programming Languages and Systems* 21.4 (1999), pp. 848–894.

[43] David Van Horn and Matthew Might. "An Analytic Framework for JavaScript". In: (2011). preprint. arXiv:`1109.4467v1 [cs.DS]`.

[44] Susan Horwitz, Alan Demers, and Tim Teitebaum. "An efficient general iterative algorithm for dataflow analysis". In: *Acta Informatica* 24.6 (1987), pp. 679–694.

[45] Susan Horwitz, Thomas Reps, and Mooly Sagiv. "Demand interprocedural dataflow analysis". In: *Proc. 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering, FSE '95*. 1995.

[46] David Hovemeyer and William Pugh. "Finding Bugs is Easy". In: *Proc. 19th Conference on Object-Oriented Programming, Systems, Languages and Applications, OOPSLA '04*. 2004, pp. 132–136.

[47] Apple Inc. *SquirrelFish Bytecodes*. URL: `http://webkit.org/specs/squirrelfish-bytecode.html`.

[48] Suresh Jagannathan, Peter Thiemann, Stephen Weeks, and Andrew Wright. "Single and loving it: Must-alias analysis for higher-order languages". In: *Proc. 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '98*. 1998, pp. 329–341.

[49] Dongseok Jang and Kwang-Moo Choe. "Points-to Analysis for JavaScript". In: *Proc. 24th Annual ACM Symposium on Applied Computing, SAC '09, Programming Language Track*. 2009.

[50] Simon Holm Jensen, Magnus Madsen, and Anders Møller. "Modeling the HTML DOM and Browser API in Static Analysis of JavaScript Web Applications". In: *Proc. 8th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. 2011.

[51] Simon Holm Jensen, Anders Møller, and Peter Thiemann. "Interprocedural Analysis with Lazy Propagation". In: *Proc. 17th International Static Analysis Symposium, SAS '10*. Vol. 6337. LNCS. Springer-Verlag, 2010, pp. 238–256.

[52] Simon Holm Jensen, Anders Møller, and Peter Thiemann. "Type Analysis for JavaScript". In: *Proc. 16th International Static Analysis Symposium (SAS)*. Vol. 5673. Springer-Verlag, 2009.

[53] S. C. Johnson. "Lint, a C Program Checker". In: *COMP. SCI. TECH. REP.* 1978, pp. 78–1273.

[54] Neil D. Jones and Steven S. Muchnick. "A flexible approach to interprocedural data flow analysis and programs with recursive data structures". In: *Proc. of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '82*. 1982, pp. 66–74.

[55] John B. Kam and Jeffrey D. Ullman. "Global Data Flow Analysis and Iterative Algorithms". In: *Journal of the ACM* 23.1 (1976), pp. 158–171.

[56] John B. Kam and Jeffrey D. Ullman. "Monotone Data Flow Analysis Frameworks". In: *Acta Informatica* 7 (1977). Springer-Verlag, pp. 305–317.

[57] R. Kelsey, W. Clinger, J. Rees, H. Abelson, N.I. Adams IV, D.H. Bartley, G. Brooks, R.K. Dybvig, D.P. Friedman, R. Halstead, et al. "Revisedˆ 5 Report on the Algorithmic Language Scheme". In: *ACM SIGPLAN Notices* (2004).

[58] Gary A. Kildall. "A Unified Approach to Global Program Optimization". In: *Proc. 1st ACM Symposium on Principles of Programming Languages, POPL '73*. 1973.

[59] Rasmus Kromann-Larsen and Rune Simonsen. "Statisk Analyse af JavaScript: Indledende arbejde". (In Danish). MA thesis. Department of Computer Science, University of Aarhus, 2007.

[60] A. Le Hors, P. Le Hégaret, L. Wood, G. Nicol, J. Robie, M. Champion, and S. Byrne. *Document object model (DOM) level 3 core specification*. 2004. URL: `http://www.w3.org/TR/DOM-Level-3-Core/`.

[61] Tobias Lindahl and Konstantinos Sagonas. "Practical type inference based on success typings". In: *Proc. 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming, PPDP '06*. 2006, pp. 167–178.

[62] Barbara Liskov and Stephen N. Zilles. "Programming with Abstract Data Types". In: *ACM SIGPLAN Notices* 9.4 (1974), pp. 50–59.

[63] Francesco Logozzo and Herman Venter. "RATA: Rapid Atomic Type Analysis by Abstract Interpretation - Application to JavaScript Optimization". In: *Proc. 19th International Conference on Compiler Construction, CC '10*. Vol. 6011. LNCS. Springer-Verlag, 2010.

[64] Sergio Maffeis, John C. Mitchell, and Ankur Taly. "An Operational Semantics for JavaScript". In: *Proc. 6th Asian Symposium on Programming Languages and Systems, APLAS '08*. Vol. 5356. LNCS. Springer-Verlag, 2008.

[65] Simon Marlow and Philip Wadler. "A practical subtyping system for Erlang". In: *Proc. 2nd ACM SIGPLAN International Conference on Functional Programming, ICFP '97*. 1997, pp. 136–149.

[66] Fadi Meawad, Gregor Richards, Floréal Morandat, and Jan Vitek. "Eval begone!: semi-automated removal of eval from javascript programs". In: *Proc of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA '12*. 2012, pp. 607–620.

[67] Sun Microsystems and Netscape Inc. *Netscape and Sun Announce Javascript(TM), the Open, Cross-Platform Object Scripting Language for Enterprise Networks and the Internet*. 1995. URL: `http://sunsite.nus.sg/hotjava/pr951204-03.html`.

[68] Matthew Might and Olin Shivers. "Improving flow analyses via ΓCFA: abstract garbage collection and counting". In: *Proc. 11th ACM SIGPLAN International Conference on Functional Programming, ICFP '06*. 2006.

[69] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. "Parameterized object sensitivity for points-to analysis for Java". In: *ACM Trans. Softw. Eng. Methodol.* 14.1 (Jan. 2005), pp. 1–41. ISSN: 1049-331X.

[70] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999. ISBN: 3540654100.

[71] Sven-Olof Nyström. "A soft-typing system for Erlang". In: *Proc. 2nd ACM SIGPLAN Erlang Workshop, ERLANG '03*. 2003, pp. 56–71.

[72] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. "Local Reasoning about Programs that Alter Data Structures". In: *Proc. 15th International Workshop on Computer Science Logic, CSL '01*. 2001.

[73] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. "Feedback-Directed Random Test Generation". In: *Proc. 29th International Conference on Software Engineering, ICSE '07*. 2007.

[74] C. Park, H. Lee, and S. Ryu. "An empirical study on the rewritability of the with statement in javascript". In: *Proc. International Workshops on Foundations of Object-Oriented Languages, FOOL '11*. 2011.

[75] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987. ISBN: 013453333X.

[76] Joe Gibbs Politz, Matthew J. Carroll, Benjamin S. Lerner, Justin Pombrio, and Shriram Krishnamurthi. In: *Proceedings of the 8th Symposium on Dynamic Languages, DLS '12*. 2012, pp. 1–16.

[77] Thomas Reps, Susan Horwitz, and Mooly Sagiv. "Precise interprocedural dataflow analysis via graph reachability". In: *Proc. 22th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '95*. 1995, pp. 49–61.

[78] Gregor Richards, Christian Hammer, Brian Burg, and Jan 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*. Vol. 6813. LNCS. 2011.

[79] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. "An analysis of the dynamic behavior of JavaScript programs". In: *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '10*. 2010.

[80] Barbara G. Ryder, William Landi, Phil Stocks, Sean Zhang, and Rita Altucher. "A schema for interprocedural modification side-effect analysis with pointer aliasing". In: *ACM Transactions on Programming Languages and Systems* 23.2 (2001), pp. 105–186.

[81] Shmuel Sagiv, Thomas W. Reps, and Susan Horwitz. "Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation". In: *Theoretical Computer Science* 167.1&2 (1996), pp. 131–170.

[82] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. "Parametric shape analysis via 3-valued logic". In: *ACM Transactions on Programming Languages and Systems* 24.3 (2002), pp. 217–298.

[83] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Stephen McCamant, Dawn Song, and Feng Mao. "A Symbolic Execution Framework for JavaScript". In: *Proc. 31st IEEE Symposium. on Security and Privacy, S&P '10*. 2010.

[84] Koushik Sen, Darko Marinov, and Gul Agha. "CUTE: a concolic unit testing engine for C". In: *Proc. 10th European Software Engineering Conference / 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE '05*. 2005.

[85] Micha Sharir and Amir Pnueli. "Two approaches to interprocedural data flow analysis". In: *Program Flow Analysis: Theory and Applications*. Ed. by Steven S. Muchnick and Neil D. Jones. Prentice-Hall, 1981. Chap. 7, pp. 189–234.

[86] Michael Sperber, R. Kent Dybvig, Matthew Flatt, and Anton van Straaten, eds. *Revised^6 Report of the Algorithmic Language Scheme – Standard Libraries*. `http://www.r6rs.org/`, 2007.

[87] Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip. "Correlation Tracking for Points-To Analysis of JavaScript". In: *Proc. of the 26th European conference on Object-oriented programming, ECOOP 12*. 2012, pp. 435–458.

[88] Peter Thiemann. "A Type Safe DOM API". In: *Proc. 10th International Workshop on Database Programming Languages, DBPL '05*. Vol. 3774. LNCS. Springer-Verlag, 2005.

[89] Peter Thiemann. "Towards a Type System for Analyzing JavaScript Programs". In: *Proc. Programming Languages and Systems, 14th European Symposium on Programming, ESOP '05*. 2005.

[90] Teck Bok Tok, Samuel Z. Guyer, and Calvin Lin. "Efficient Flow-Sensitive Interprocedural Data-Flow Analysis in the Presence of Pointers". In: *Proc. 15th International Conference on Compiler Construction, CC '06*. 2006, pp. 17–31.

[91] David Ungar and Randall B Smith. "Self: The power of simplicity". In: *SIGPLAN Notices* 22.12 (1987), pp. 227–242. ISSN: 0362-1340.

[92] Andrew K. Wright and Robert Cartwright. "A Practical Soft Type System for Scheme". In: *ACM Transactions on Programming Languages and Systems* 19.1 (1997), pp. 87–152.

[93] Yichen Xie and Alex Aiken. "Static Detection of Security Vulnerabilities in Scripting Languages". In: *Proc. 15th USENIX Security Symposium*. 2006.

[94] Tian Zhao. "Polymorphic type inference for scripting languages with object extensions". In: *Proceedings of the 7th Symposium on Dynamic Languages, DLS 11*. 2011, pp. 37–50.