Improving Tools for JavaScript Programmers

(Position Paper)

Esben Andreasen
Aarhus University
esbena@cs.au.dk

Asger Feldthaus
Aarhus University
asf@cs.au.dk

Simon Holm Jensen
Aarhus University
simonhj@cs.au.dk

Casper S. Jensen
Aarhus University
semadk@cs.au.dk

Peter A. Jonsson
Aarhus University
pjonsson@cs.au.dk

Magnus Madsen
Aarhus University
magnusm@cs.au.dk

Anders Møller
Aarhus University
amoeller@cs.au.dk

ABSTRACT

We present an overview of three research projects that all aim to provide better tools for JavaScript web application programmers[1]: *TAJS*, which infers static type information for JavaScript applications using dataflow analysis; *JSRefactor*, which enables sound code refactorings; and *Artemis*, which provides high-coverage automated testing.

1. JAVASCRIPT PROGRAMMERS NEED BETTER TOOLS

JavaScript contains many dynamic features that allegedly ease the task of programming modern web applications. Most importantly, it has a flexible notion of objects: properties are added dynamically, the names of the properties are dynamically computed strings, the types of the properties are not fixed, and prototype relations between objects change during execution. An experimental study has shown that most dynamic features in JavaScript are widely used [10].

Such flexibility has a price. It becomes challenging to reason about the behavior of JavaScript programs without actually running them. To make matters worse, the language provides no encapsulation mechanisms, except for local variables in closures. For many kinds of programming errors that cause compilation errors or runtime errors in other languages, JavaScript programs keep on running, often with surprising consequences.

As a consequence, JavaScript programmers must rely on tedious testing to a much greater extent than necessary with statically typed languages. Additionally, it is difficult to foresee the consequences of modifications to the code, so code refactoring is rarely applied. Unlike the first scripts that appeared when JavaScript was introduced, today's JavaScript programs often contain thousands of lines of code, so it becomes increasingly important to develop better tool support for the JavaScript programmers.

[1]For more information about the projects and tools, see the website for Center for Advanced Software Analysis at `http://cs.au.dk/CASA`.

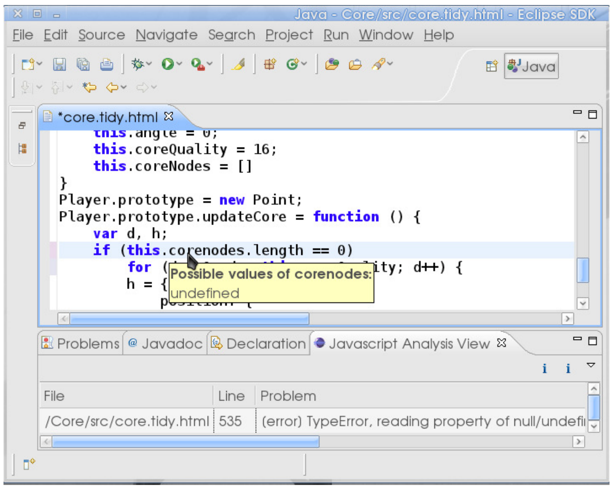
2. FINDING ERRORS WITH DATAFLOW ANALYSIS

The TAJS analysis tool infers an abstract state for each program point in a given JavaScript web application. Such an abstract state soundly models the possible states that may appear at runtime and can be used for detecting type-related errors and dead code. These errors often arise from wrong function parameters, misunderstandings of the runtime type coercion rules, or simple typos that can be tedious to find using testing.

We have approached the development of TAJS in stages. First, our focus has been on the abstract domain and dataflow constraints that are required for a sound and reasonably precise modeling of the basic operations of the JavaScript language itself and the native objects that are specified in the ECMAScript standard [3]. This involves an extraordinarily elaborate lattice structure and models of the intricate details of identifier and object property resolution, prototype chains, property attributes, scope chains, type coercions, etc. [7]. The resulting static analysis is flow- and partially context-sensitive. It performs constant propagation for primitive values and models object references using recency abstraction. For every expression, the analysis provides an over-approximation of its possible runtime types and values, which can be analyzed subsequently to detect likely errors.

Next, to reason about web application code, we also need to model the browser API, including the HTML DOM and the event system, with involves additional hundreds of objects, functions, and properties [6]. In parallel, we have developed new techniques for interprocedural dataflow analysis to boost performance. Our *lazy propagation* technique is particularly suitable for the large abstract states that we encounter [8]. More recently, we have taken the first step of handling common patterns of code that is dynamically generated using the `eval` function [5], using the study by Richards et al. [9] as a starting point.

Altogether, these techniques enable analysis of JavaScript web applications up to a few thousand lines of code, although the scalability is naturally highly affected by the complexity of the code. We have demonstrated that the approach can infer type information and call graphs with good precision


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

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

Possible values of corenodes:
undefined

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

Figure 1: The TAJS analysis plug-in for Eclipse, reporting a programming error and highlighting the type inferred for the selected expression [6].

and provide useful warning messages when type-related errors occur. We envision such information being made available to the programmer during development; a screenshot from our prototype plugin for Eclipse is shown in Figure 1.

Our current work focuses on improving the analysis performance. As the average JavaScript programs become larger and often involve libraries, it becomes increasingly important that the scalability of the analysis is improved. Specifically, we are studying the performance bottlenecks that appear with applications that use jQuery, using the idea of correlation tracking that has recently been proposed by Sridharan et al. [11].

3. TOOL-SUPPORTED REFACTORING

Refactoring is a popular technique for improving the structure of programs while preserving their behavior. Tool support is indispensable for finding the necessary changes when the programmer suggests a specific refactoring and for ensuring that the program behavior is preserved. However, refactoring tools for JavaScript cannot use the techniques that have been developed for e.g. Java since they rely on information about static types and class hierarchies. As an example, no existing mainstream JavaScript IDE can perform even apparently simple refactorings, such as, renaming an object property, in a sound and precise manner.

In the JSRefactor project, we explore the use of pointer analysis as a foundation for providing better tool support for refactoring for JavaScript programs [4]. As a starting point we consider renaming of variables or object properties, but also more JavaScript-specific refactorings – encapsulation of properties and extraction of modules – that target programming idioms advocated by influential practitioners [2].

Beside supporting additional refactorings in our framework, an important next step is to improve the scalability of the underlying pointer analysis. On the theoretical side, it remains an interesting challenge how to ensure that the refactoring specifications we provide are sound with respect to the semantics of JavaScript.

4. AUTOMATED TESTING

Testing JavaScript web applications is tedious but necessary. The goal of the Artemis project is to automate the production of high-coverage test inputs [1]. This can be seen as a complementary approach to TAJS. Although testing cannot show absence of errors – in contrast to the static analysis approach we use in TAJS – one may argue that dynamic approaches to error detection are better suited for dynamic languages like JavaScript. As a case in point, `eval` causes no complications in Artemis, unlike in TAJS.

The approach we take in Artemis is to apply light-weight feedback-directed random testing. A test input consists of a sequence of parameterized events that trigger execution of code. The Artemis tool monitors the execution to collect information that suggests promising new inputs that may improve coverage.

Our first version of Artemis was based on Envjs, which is a simulated browser environment written in JavaScript, and included various heuristics for generating and prioritizing new inputs. We are currently integrating our algorithms into the more robust WebKit infrastructure and exploring more powerful heuristics for providing higher and faster coverage of typical JavaScript applications.

Acknowledgments

We appreciate the contributions to the TAJS, JSRefactor, and Artemis projects by Kristoffer Just Andersen, Shay Artzi, Julian Dolby, Matthias Diehn Ingesman, Jacob H.C. Kragh, Todd Millstein, Max Schäfer, Peter Thiemann, and Frank Tip. This work was supported by Google, IBM, and The Danish Research Council for Technology and Production.

5. REFERENCES

[1] S. Artzi, J. Dolby, S. H. Jensen, A. Møller, and F. Tip. A framework for automated testing of JavaScript web applications. In *ICSE'11*, May 2011.

[2] D. Crockford. *JavaScript: The Good Parts*. O'Reilly, 2008.

[3] ECMA. ECMAScript Language Specification, 3rd edition, 2000. ECMA-262.

[4] A. Feldthaus, T. Millstein, A. Møller, M. Schäfer, and F. Tip. Tool-supported refactoring for JavaScript. In *OOPSLA'11*, October 2011.

[5] S. H. Jensen, P. A. Jonsson, and A. Møller. Remedying the eval that men do. In *ISSTA'12*, July 2012.

[6] S. H. Jensen, M. Madsen, and A. Møller. Modeling the HTML DOM and browser API in static analysis of JavaScript web applications. In *ESEC/FSE'11*, September 2011.

[7] S. H. Jensen, A. Møller, and P. Thiemann. Type analysis for JavaScript. In *SAS'09*, August 2009.

[8] S. H. Jensen, A. Møller, and P. Thiemann. Interprocedural analysis with lazy propagation. In *SAS'10*, September 2010.

[9] G. Richards, C. Hammer, B. Burg, and J. Vitek. The eval that men do - a large-scale study of the use of eval in JavaScript applications. In *ECOOP'11*, July 2011.

[10] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of Javascript programs. In *PLDI'10*, June 2010.

[11] M. Sridharan, J. Dolby, S. Chandra, M. Schäfer, and F. Tip. Correlation tracking for points-to analysis of JavaScript. In *ECOOP'12*, June 2012.