Static Checking of Range Assertions in JavaScript Programs

Astrid Younang and Lunjin Lu

***Abstract*—Because of the dynamic nature of JavaScript, an array access operation with a property (index) that is out of its range will not throw an arrayIndexoutOfBound exception, but will silently return the value undefined. This can cause programs to crash or malfunction. This work extends the JavaScript language with range assertions and allows developers to insert them at any program point. Range assertions could help detect such silent arrayIndexoutOfBound exceptions and can be useful for program understanding and debugging. We propose an assertion language that can be used in any JavaScript static analyzer. Assertions are statically checked and possible violations are reported. The experiments on a set of benchmark programs reported a violation that would have been previously unnoticed.**

***Index Terms*—Abstract interpretation, interval domain, JavaScript, range assertions.**

I. Introduction

Static analysis, which is the automatic discovery of program properties, has long been used for program understanding and program verification. Program developers rely on it to identify potential errors and correct them. For programming languages like C and Java, state of the art static analyzers are available for each stage of development thanks to their static nature. However, the situation is different for JavaScript. It supports first class functions, uses prototype inheritance and dynamic typing. The very same features that made JavaScript special, easy to use and attractive for developers are the same ones behind the difficulty of developing static analyzers that are scalable and precise enough.

With the increasing use of the language, a lot of effort has been done this last decade by the research community to equip JavaScript developers with better tools. Contributions have been made on pointer analysis [1], type inference analysis [2]-[7] and vulnerabilities detection [8]-[12]. They include static, dynamic or blended analysis approaches for the whole language or just a subset. This paper focuses solely on abstract interpretation based static analysis. Abstract interpretation is a theory of semantic based approximations formalized by Cousot and Cousot [13], [14]. Often required for the analysis of large and complex programs, it allows the analysis of a program in an abstract universe equipped with abstract domains.

Several abstract interpretation-based static analyzers have been proposed for JavaScript programs. Those static analyzers detect and report various bugs in JavaScript programs such as type errors, reference errors, null/undefined variables and unreachable code. TAJS is a tool that detects type related errors in JavaScript programs. TAJS was designed and implemented by Jensen *et al.* [4]. It tracks undefinedness, nullness, type and point-to information and uses recency abstraction to increase analysis precision. TAJS detects definite type errors in JavaScript programs and generates warnings on potential type errors. Kashyap *et al.* [15] designed and implemented JSAI (JavaScript Abstract Interpreter), an abstract interpreter for static analysis of JavaScript programs. JSAI, which has a similar purpose as TAJS, detects and reports type errors in JavaScript programs. The main difference between the two tools is context sensitivity. In JSAI, the user can choose between a range of context-sensitivities. Lee *et al*. proposed SAFE [16], a Scalable Analysis Framework for ECMAScript. It provides three intermediate representations for JavaScript that can be used in various analyses and optimizations. SAFE detects range errors, reference errors, syntax errors, type and URI errors and successfully generates warnings such as the reading of an absent property of an object or a conditional expression that is always true or false. Range assertions can be viewed as an additional feature to be included in current static analyzers.

The main contributions of this paper are:

1. The extension of the JavaScript language with range assertions. We designed an assertion language that can be integrated in any JavaScript static analyzer.
2. An empirical evaluation of assertion checks on a set of benchmarks. Range assertions were manually inserted in the analyzed programs. On most of the benchmarks, the inserted assertions were satisfied, which is a sign of a good execution of the programs as far as the checked properties were concerned. Some assertions were reported to have failed due to the over-approximation of the analysis. We found one real violation on 7 benchmark programs analyzed.

II. Motivating Example

Arrays in JavaScript are different from regular objects with the *length* property. An index is a string of digit characters representing a positive integer property between 0 and 2^{32}-2. The *length* property is automatically updated to the maximum index plus 1. Due to its dynamic nature, JavaScript allows array objects to have properties that are negative numbers, non-integer numbers and non-numeric values. When accessing a non-existing property in an array, JavaScript does not throw an *arrayIndexoutOfBound*

Manuscript received February 23, 2017; revised April 25, 2017. This work was supported by Oakland University.

Lunjin Lu and Astrid Younang are with the Computer Science and Engineering department at Oakland University in Michigan, USA (e-mail: L2Lu@ oakland.edu, awaindja@oakland.edu).

DOI: 10.7763/IJCTE.2017.V9.1164

exception, but silently returns the value *undefined*. Range assertions could be useful as to check the range of array properties and identify such silent *arrayIndexoutOfBound* errors. The example below illustrates a range assertion check in a JavaScript program.

```
1  var i,number,ind=0;
2  var A=[1,2,3,4,5,6,7,8,9,10];
3
4  while (ind<7) {
5   i=i+3;
6   assert("range", i,0,9); // range assertion
7   number=A[i];
8   f(number,i);
9   i=i+ind;
10      ind=ind+2;
11      }
12
13 function f(x,y){
14      var result;
15  result=x.toPrecision(y);
16      }
```

In the above example, the range assertion fails because the variable *i* has not been initialized. The JavaScript engine will continue the execution of the program with *undefined* as the value of *i* and this will impact the rest of the program. The condition in line 4 successfully evaluates to true at the first iteration. The value of *i* in line 5 is *undefined*, the assertion in line 6 fails and the array access in line 7 fails silently. A regular execution without assertions will not throw errors. The failed assertion in line 6 comes as a warning that this will compromise the following array access operation. The program then calls the function *f* in line 8 with 2 arguments that are *undefined*. At the entrance of the function, there is a possible type error that can occur in line 15 as the program cannot call the method *toPrecision* of *undefined*. This example shows that the insertion of assertions in the program can detect errors that would otherwise occur silently. This is an example where an initialization that has been forgotten has compromised the execution of the program. Due to the lack of static type checking in JavaScript programs, program developers can encounter the proliferation of *undefined* values throughout their programs. Also, in the presence of boolean expressions in loops or conditional statements, an *undefined* value will cause the boolean expression to always evaluate to false as the *undefined* value is converted to *NaN*.

III. The Range Assertion Language

In this section, we present an assertion language for JavaScript. Let *Number* be the set of numbers, *Variable* the set of variables and *Stmt* the set of statements. *Number* consists of 64-bit floating point numbers as defined by the IEEE-754 standard [17]. Assertion statements inserted in JavaScript programs respect the following grammar:

$$\begin{gathered} n1, n2 \in Number, x \in Variable \\ s \in Stmt ::= ... \mid assert(range, v, n1, n2); \mid ... \\ v ::= x \mid v.x \end{gathered} \tag{1}$$

Fig. 1 illustrates the JavaScript Abstract Interpreter augmented with assertions. Before a JavaScript program is analyzed. It is passed to the Mozilla Rhino JavaScript parser to produce a Rhino abstract syntax tree (AST) [18]. The Rhino AST is then passed to a translator to produce another abstract syntax tree called notJS. The translator was modified to recognize assertion statements and produces a new node in the notJS intermediate representation. The assertion statement is a special function call. All the details about the translator can be found in [15]. The static analysis engine takes as input the abstract syntax tree extended with assertion statements.

An abstract state $\hat{S}$ is a tuple of 4 elements. The first element $\hat{t}$ is the next statement to execute, followed by the abstract environment $\hat{\rho}$ which is a map from variables to locations. The abstract store $\hat{\sigma}$ is a map from locations to values and the continuation stack k is the stack that contains the rest of the statements to execute to reach the final state. The semantic rule for the assertions statements is as follows:

$$\frac{\hat{S}_{in} =< s_r, \hat{\rho}, \hat{\sigma}, k >,\ k = s :: k'}{\hat{S}_{out} =< s, \hat{\rho}, \hat{\sigma}, k' >} \qquad s_r = assert(range, v, n_1, n_2) \tag{2}$$

The statement s_r represents a range assertion statement. The assertions statements do not modify the environment nor the store. Let b_l and b_u be two abstract boolean variables.

$$b_l = \begin{cases} True & \text{if } n_1 \leq \hat{\sigma}(\hat{\rho}(v)) \\ False & \text{if } \hat{\sigma}(\hat{\rho}(v)) < n_1 \\ \top_{\mathbb{B}} & \text{otherwise} \end{cases} \tag{3}$$

$$b_u = \begin{cases} True & \text{if } \hat{\sigma}(\hat{\rho}(v)) \leq n_2 \\ False & \text{if } n_2 < \hat{\sigma}(\hat{\rho}(v)) \\ \top_{\mathbb{B}} & \text{otherwise} \end{cases} \tag{4}$$

The variable b_l evaluates to *True* when the value of the variable *v* is greater or equal to n_1, *False* when it is strictly smaller than n_1 and T_B when we cannot precisely compare the values due to the imprecision of the analysis. The same reasoning applies to the boolean variable b_u.

The abstract numeric domain used in our analysis is the extended abstract domain of intervals from our previous work in [19]. Let *Float754* denote the set of all IEEE-754 numbers including the special numbers *NaN*,$+\infty$ and $-\infty$. The extended abstract domain of intervals $I^\sharp$ is defined as follows:

$$\begin{aligned} I^\sharp = &\{\top, \bot, NaN, Int32\} \\ &\cup \{NConst(a) \mid a \in Float754 \setminus \{NaN, +\infty, -\infty\}\} \\ &\cup \{Int(a,b), Norm(a,b) \mid (a, b \in \mathbb{Z}_{\mathbf{s}}^{\mathbf{b}}) \wedge a \leq b\} \end{aligned} \tag{5}$$

$$\mathbb{Z}_{\mathbf{s}}^{\mathbf{b}} = \{n \mid n \in \mathbb{Z} \wedge \mathbf{s} \leq n \leq \mathbf{b}\} \cup \{+\infty, -\infty\} \tag{6}$$

Norm (*a*,*b*) describes the set of real numbers between *a* and *b* including *NaN*, *Int*(*a*,*b*) the same set of real numbers without *NaN*. *NConst*(*c*) describes the real number *c* and

Int32 the set of all unsigned 32-bit integers. *NaN* is the special number arising from computations such as $\infty \times 0$ or $0 \div 0$.

The concretization function $\gamma : I^\sharp \longrightarrow \mathbb{P}(Float754)$ for the extended abstract domain of intervals is defined as follows:

$$
\begin{array}{rcl}
\gamma(\bot) & = & \{\} \\
\gamma(NaN) & = & \{NaN\} \\
\gamma(Norm(a,b)) & = & \{r | r \in Float754, a \le r \le b\} \cup \{NaN\} \\
\gamma(Int(a,b)) & = & \{r | r \in Float754, a \le r \le b\} \\
\gamma(Int32) & = & \{r | r \in \mathbb{Z}, 0 \le r \le 2^{32}-1\} \\
\gamma(NConst(a)) & = & \{a\}\ for\ a \in Float754 \setminus \{NaN, +\infty, -\infty\} \\
\gamma(\top) & = & Float754
\end{array}
\qquad (7)
$$

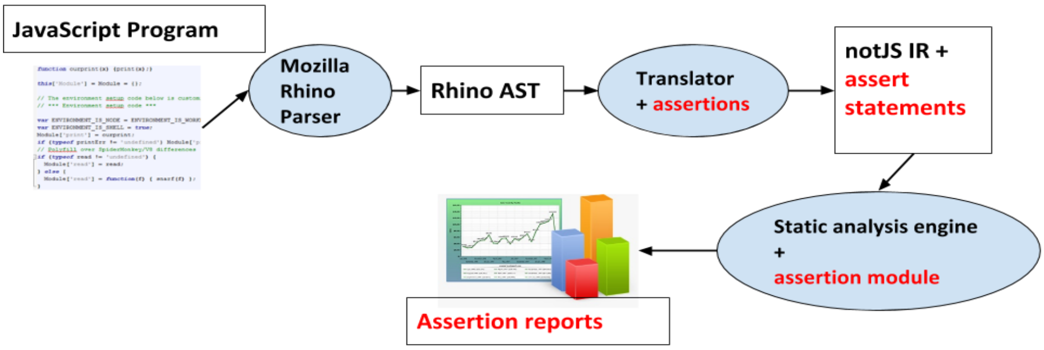

Fig. 1. JavaScript abstract interpreter (JSAI) augmented with assertions.

IV. Approximation of Math Functions

Each built-in mathematical function *f* over the domain of numbers is simulated by a corresponding abstract function f# over the domain of intervals. Abstract mathematical functions such as *abs#, acos#, asin#, atan#, atan2#, ceil#, cos#, exp#, floor#, log#, min#, max#, pow#, random#, round#, sin#, sqrt#, tan#* have been designed and implemented in the analyzer using the interval domain. We now give definitions for some of these abstract functions.

$$
log^\sharp(Int(a,b)) = \begin{cases} N\hat{a}N & \text{if } (b<0) \\ Int(-\infty,-\infty) & \text{if } (a=0 \wedge b=0) \\ Int(0,0) & \text{if } (a=1 \wedge b=1) \\ Int(+\infty,+\infty) & \text{if } (a=+\infty \wedge b=+\infty) \\ Int(\llcorner log(a) \lrcorner, \ulcorner log(b) \urcorner) & \text{if } (a>0) \\ Int(-\infty, \ulcorner log(b) \urcorner) & \text{if } (a \le 0 \wedge b>0) \end{cases} \qquad (8)
$$

$$
sqrt^\sharp(Int(a,b)) = \begin{cases} N\hat{a}N & \text{if } (b<0) \\ Norm(0, \ulcorner sqrt(b) \urcorner) & \text{if } (a \le 0 \wedge b \ge 0) \\ Int(\llcorner sqrt(a) \lrcorner, \ulcorner sqrt(b) \urcorner) & \text{otherwise} \end{cases} \qquad (9)
$$

$$
abs^\sharp(Int(a,b)) = \begin{cases} Int(a,b) & \text{if } (a \ge 0 \wedge b \ge 0) \\ Int(0, max(-a,b)) & \text{if } (a \le 0 \wedge b \ge 0) \\ Int(-b,-a) & \text{if } (a \le 0 \wedge b \le 0) \end{cases} \qquad (10)
$$

$$
atan2^\sharp(NConst(c), Norm(a,b)) = \begin{cases} Norm(1,2) & \text{if } (c>0 \wedge a=0 \wedge b=0) \\ Norm(0,0) & \text{if } (c=0 \wedge a>0) \\ Norm(0,0) & \text{if } (c=0 \wedge a=0 \wedge b=0) \\ Norm(-2,-1) & \text{if } (c<0 \wedge a=0 \wedge b=0) \\ Norm(-4,4) & \text{otherwise} \end{cases} \qquad (11)
$$

The abstract mathematical functions were defined for all the elements of the extended interval domain. Above are the definitions for the *log#, sqrt#, abs# and atan2#* on abstract elements *Int (a,b), NConst(c) and Norm(a,b).* The function artan2 () returns the arctangent of the quotient of its arguments which is a value between -π and π. Depending on the intervals, different results are obtained when computing their arctangent. Similarly, the absolute value of an interval can be computed based on the values of its bounds.

V. Evaluation

We used the abstract interpreter JSAI from [15] to test the assertions. JSAI is a framework written in Scala. We ran JSAI on a Scientific Linux 6.3 distribution with 24 Intel Xeon CPUs with a capacity of 1.6GHz and 32GB memory. The modifications made to JSAI are detailed in Section III. The benchmarks chosen are the standard SunSpider [21] and V8 programs [22], browser addon programs from the Mozilla addon repository [23], machine generated JavaScript code from the Emscripten LLVM test suite [24].

We used the following rules to insert range assertions in our programs:

- If the length of an array is known, then we insert a range assertion statement before any array access with the lower bound equals to 0 and the upper bound equals to the value of the length
- If the length of an array is unknown, we insert a range assertion statement with the lower bound equals to 0 and the upper bound equals to the value of the maximum index in a JavaScript array which is 4294967295.
- Some programs like *ems-aha.js* had their own assertion statements like $assert(i >= 0\ \&\&\ i < FS.streams.length);$
- We use those statements to add additional range assertions statements.

A. Precision

The precision metric used in our benchmark programs for range assertions is the number of program points satisfying or violating an assertion. Fig. 2 presents the reports on the benchmark programs. The source codes of the first set of benchmark programs with the prefix adn are similar. They all reported the same program point for the violation of the range assertion. There is an access operation on the array *globalFuncList[TopNum]* with *TopNum=Date.now();*

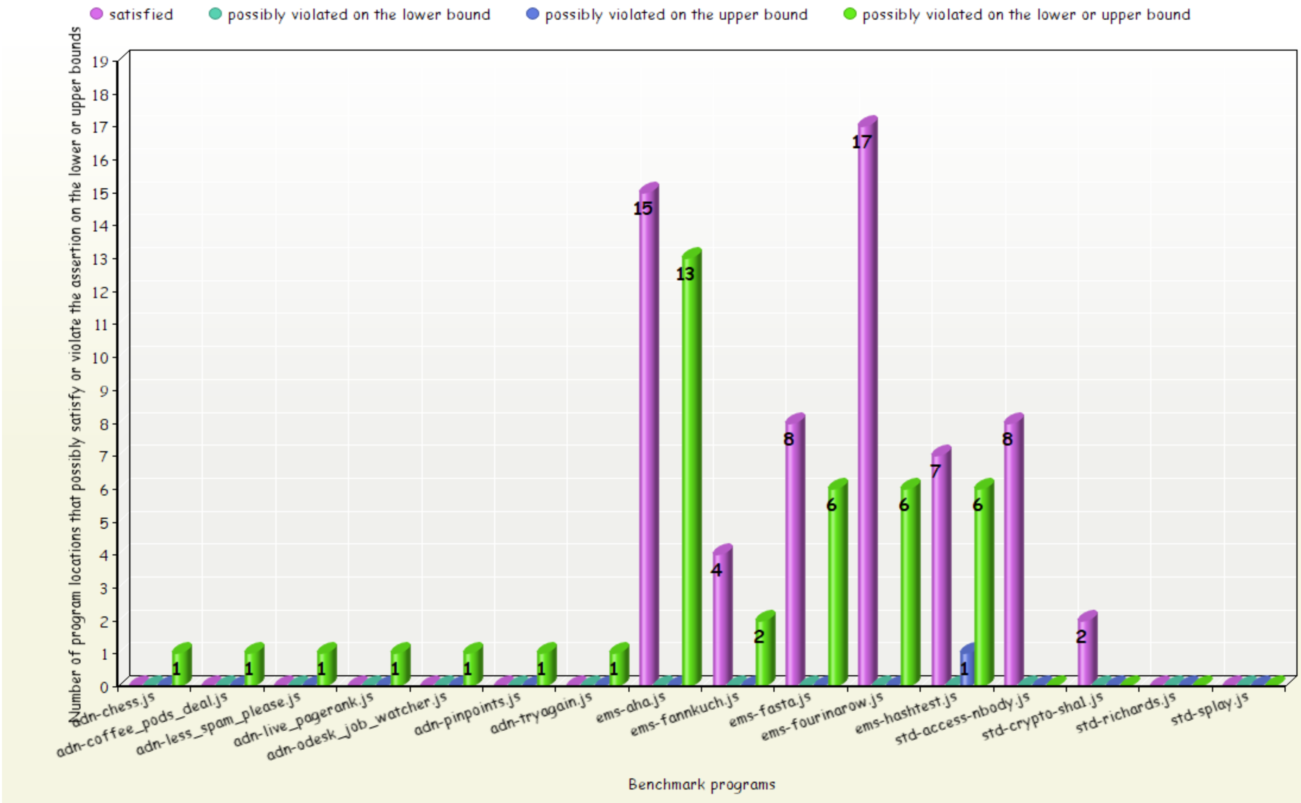

Fig. 2. Report on range assertions. For each benchmark program, we count the number of programs locations that possibly satisfy or violate the assertions on the lower or upper bounds.

Date.now() is a JavaScript function returning the number of milliseconds between midnight, January 1, 1970 and the current date and time. This value could possibly be larger than the maximum value of an array index. The possibly violated range assertions in *ems-aha.js*, *ems-fannkuch.js*, *ems-fasta.js*, *ems-fourinarow.js* and *ems-hashtest.js* are the result of the over-approximation on the bitwise shift operators. Those operators produce a number that is approximated to the abstract element *Int32*, which describes the set of unsigned 32-bit integers. Therefore, the range assertions cannot be precisely checked. This opens room for the use of a more precise numeric domain in future work. The programs *std-richards.js* and *std-splay.js* do not contain arrays.

B. *Performance*

A comparison between the running time of the analyzer with and without assertions shows no significant increase. Therefore, a JavaScript static analyzer can include assertion checks with little to no cost.

VI. Related Work

Compared to Java and C, JavaScript is still in need sophisticated tools to aid program developers in their testing activities. Sound and unsound approaches have been proposed in [8]-[12] to detect security vulnerabilities in browser extensions and JavaScript web applications. Due to the dynamic typing of JavaScript, type inference analysis has received a lot of attention [2]-[4], [6] and [7]. Contributions have also been made on pointer analysis in [1], which can be used for further JavaScript analyses. Some effort has been done by the research community and some tools have been proposed over the years to aid developers improve the quality of the JavaScript code they are writing. In this section, we focus on the main static analyzers available and the functionalities that they offer.

A. *Code Quality Tools*

JSlint, JSHint, ESLint and Closure-Linter are mainly code quality tools. They analyze JavaScript programs and report bugs based on a set of predefined rules. JSLint [25] is a code quality tool that was originally developped by Douglas Crockford from Yahoo. The tool is made available via an online interface (www.jslint.com) where a user can paste JavaScript code to be analyzed. JSlint analyzes the program over a set of strict rules and produces a report about the errors detected. In order to loosen some rules and reduce the number of errors, JSLint presents several options such as tolerate eval, tolerate messy white space and tolerate unused parameters. It also allows the program to be analyzed in different contexts by assuming different environments such as NodeJS, couchDB and ES6.

JSHint [26] was created by forking the original JSLint. The motivation behind the creation of JSHint was to allow more configuration over the options available in JSLint and to give more power to the user. Another reason behind the creation of JSHint was to reduce the number of format related errors and to focus more on errors that will cause the program to malfunction.

ESLint [27] is another tool that can be used to validate JavaScript and check for errors. It allows the user to write their own linting rules and it is designed to have all the rules completely pluggable.

Google Closure Linter [28] is another utility that can be used to check JavaScript files for issues such as missing columns or spacing. The tool follows the Google JavaScript style guide and the user has no control over the rules. Among the errors detected by those linter tools, we can cite missing semi-columns, already defined variables, null/undefined variables, use of eval function, variables that are used before they are defined.

B. Type Errors Reporting Tools

Variables in JavaScript can hold different types during the execution of a program. This leads to type errors that can cause the program to malfunction or terminate. Several frameworks are available to detect type related errors in JavaScript programs. Jensen et al in [4] introduced TAJS - Type Analysis tool for JavaScript. TAJS detects type related errors in JavaScript programs in addition to other errors such as unreachable code, reference errors, null/undefined variables, unused variables and properties that are never read. TAJS has evolved over the years and improved its precision with techniques such as recency abstraction and lazy propagation. JSAI is a JavaScript Abstract Interpreter developed by Kashyap et al in [15]. JSAI detects type and range errors in JavaScript programs. It is different from TAJS with the context sensitivity aspect which is entirely configurable by the user. SAFE is another static analyzer for JavaScript programs [16]. Unlike TAJS and JSAI which use a parser based on EcmaScript 3, SAFE is based on EcmaScript 5. It also detects type related errors in JavaScript programs in addition to reference errors, properties never read, unused variables, range errors, conditional expressions always true or false and reading of absent properties.

VII. Conclusion and Future Work

We extended the JavaScript language to support range assertions. Those assertions are statically checked in order to locate possible silent *arrayIndexOutOfBound* exceptions that could cause programs to crash or malfunctions. However, a thorough check on those assertions requires the use of sophisticated and precise abstract numeric domain. Future work will investigate the tradeoff between cost and precision of the octagon domain as abstract numeric domain.

References

[1] J. S. D. Jang and K.-M. Choe, "Points-to analysis for javascript," in *Proc. the 2009 ACM symposium on Applied Computing*, 2009, pp. 1930–1937.

[2] P. Thiemann, "Towards a type system for analyzing javascript programs," *Programming Languages and Systems*, pp. 408–422, 2005.

[3] P. Heidegger and P. Thiemann, "Recency types for analyzing scripting languages," in *Proc. ECOOP 2010–Object-Oriented Programming*, 2010, pp. 200–224.

[4] S. H. Jensen, A. Møller, and P. Thiemann, "Type analysis for javascript," *Static Analysis*, pp. 238–255, 2009.

[5] S. H. Jensen, A. Møller, and P. Thiemann, "Interprocedural analysis with lazy propagation," *Static Analysis*, pp. 320–339, 2011.

[6] C. Anderson, P. Giannini, and S. Drossopoulou, "Towards type inference for javascript," in *Proc. ECOOP 2005-Object-Oriented Programming*, 2005, pp. 428–452.

[7] F. Logozzo and H. Venter, "Rata: Rapid atomic type analysis by abstract interpretation–application to javascript optimization," *Compiler Construction*, pp. 66–83, 2010.

[8] A. Taly *et al*., "Automated analysis of security-critical javascript apis," in *Proc. 2011 IEEE Symposium on Security and Privacy (SP)*, 2011, pp. 363–378.

[9] S. Bandhakavi *et al.*, "Communications of the ACM 54," 2011.

[10] S. Guarnieri and V. B. Livshits, "Gatekeeper: Mostly static enforcement of security and reliability policies for javascript code," presented at USENIX Security Symposium, pp. 151–168, 2009.

[11] R. N. Taylor, D. L. Levine, and C. D. Kelly, "Software engineering," *IEEE Transactions on 18*, vol. 206, 1992.

[12] A. Guha, S. Krishnamurthi, and T. Jim, "Using static analysis for ajax intrusion detection," *in Proc. the 18th International Conference on World Wide Web*, 2009, pp. 561–570.

[13] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," *in Proc. the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1977, pp. 238–252.

[14] P. Cousot and R. Cousot, "Systematic design of program analysis frameworks," *in Proc. the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1979, pp. 269–282.

[15] V. Kashyap *et al.*, "Jsai: A static analysis platform for javascript," in *Proc. the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 121–132.

[16] A. H. Lee, S. Won, J. Jin, J. Cho, and S. Ryu, "Safe: Formal specification and implementation of a scalable analysis framework for ecmascript," in *Proc. International Workshop on Foundations of Object-Oriented Languages (FOOL)*, 2012.

[17] D. Zuras *et al*., *IEEE Std 754-2008*, 2008.

[18] Public class parser. [Online]. Available: http://mozilla.github.io/rhino/javadoc/org/mozilla/javascript/Parser.html

[19] A. Younang and L. Lu, "Improving precision of java script program analysis with an extended domain of intervals," in *Proc. 39th Annual Computer Software and Applications Conference, IEEE COMPSAC*, 2015, pp. 441–446.

[20] A. Cortesi and M. Zanioli, "Computer languages, systems & structures," vol. 37, no. 24, 2011.

[21] SunSpider 1.0.2 Javascript benchmark. [Online]. Available: http://webkit.org/perf/sunspider/sunspider.html

[22] Benchmarks. [Online]. Available: http://v8.google.com/svn/data/benchmarks/v7/run.html

[23] Addons. [Online]. Available: https://addons.mozilla.org/en-US/firefox/

[24] Emscripten. [Online]. Available: http://www.emscripten.org/

[25] D. Crockford. (2011). URL. [Online]. Available: http://www.jslint.com.

[26] A. Kovalyov, W. Kluge, and J. Perez, "Jshint, a javascript code quality tool," 2010.

[27] Eslint. [Online]. Available: http://eslint.org/

[28] Developers. [Online]. Available: https://developers.google.com/closure

Astrid Younang is a Ph.D candidate in the Computer Science and Engineering Department at Oakland University in Michigan. She was born in Cameroon. She earned a B.S. degree in electronics and computer engineering and a M.S. degree in computer engineering from ESIGELEC in France. She worked for three years at France Telecom for the Device in Life Management department. Her research interest is static analysis of JavaScript web applications for bug detection.

Lunjin Lu is an associate professor and Chair of the Computer Science and Engineering department at Oakland University in Michigan. His work on semantic-based program analysis is based on the abstract interpretation approach. Since 1990, he has been working on semantics-based program analysis and its applications to logic programs. The topics he has worked on range from general analysis framework to new program analysis to efficient implementation.