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 and vulnerability 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.

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).

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

```
    n1, n2 ∈ Number, x ∈ Variable
    s ∈ Stmt ::= ... | assert(range, v, n1, n2); | ...  (1)
    v ::= x | v.x
```

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 \( \hat{f} \) 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:

\[
\begin{align*}
\hat{S}_{in} & = < s_r, \hat{\rho}, \hat{\sigma}, k >, k = s :: k' \\
\hat{S}_{out} & = < s, \hat{\rho}, \hat{\sigma}, k' > \\

e & = \text{assert}(\text{range}, v, n_1, n_2)
\end{align*}
\]

The state \( s_e \) 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.

\[
\begin{align*}
b_l & = \begin{cases} 
True & \text{if } n_1 \leq \hat{\sigma}(\hat{\rho}(v)) \\
False & \text{if } \hat{\sigma}(\hat{\rho}(v)) < n_1 \\
T_B & \text{otherwise}
\end{cases} \\
b_u & = \begin{cases} 
True & \text{if } \hat{\sigma}(\hat{\rho}(v)) \leq n_2 \\
False & \text{if } n_2 < \hat{\sigma}(\hat{\rho}(v)) \\
T_B & \text{otherwise}
\end{cases}
\end{align*}
\]

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`, `+∞` and `−∞`. The extended abstract domain of intervals \( I^s \) is defined as follows:

\[
I^s = \{ T, ⊥, NaN, Int32 \} \\
\cup \{ \text{NConst}(a) \mid a ∈ \text{Float754} \setminus \{ NaN, +∞, −∞ \} \} \\
\cup \{ \text{Int}(a, b), \text{Norm}(a, b) \mid (a, b) ∈ Z^b_a \land a ≤ b \} \\
\cup \{ n \mid n ∈ \mathbb{Z} \land s ≤ n ≤ b \} ∪ \{ +∞, −∞ \}
\]

`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
The concretization function $\gamma : I^2 \rightarrow P(\text{Float754})$ for the extended abstract domain of intervals is defined as follows:

$$
\begin{align*}
\gamma(\mathbb{N}_+) &= \{1\} \\
\gamma(\mathbb{N}_N) &= \{\mathbb{N}_N\} \\
\gamma(\text{Norm}(a,b)) &= \{r | r \in \text{Float754}, a \leq r \leq b \cup \{\mathbb{N}_N\} \\
\gamma(\text{Int}(a,b)) &= \{r | r \in \text{Float754}, a \leq r \leq b \} \\
\gamma(\text{Int}(0)) &= \{0\} \\
\gamma(\text{Int}(0)) &= \{0\} \\
\gamma(\text{Norm}(a,b)) &= \{0, 1, \ldots, \max(2^{32} - 1, b-a)\} \\
\gamma(\text{Int}(0)) &= \{\mathbb{N}_N\} \\
\gamma(\text{Int}(0)) &= \{\mathbb{N}_N\} \\
\gamma(\text{Int}(0)) &= \{\mathbb{N}_N\} \\
\text{atan2}(\text{Norm}(a), \text{Norm}(b)) &= \text{atan2}(\text{Norm}(a), \text{Norm}(b)) \\
\text{atan2}(\text{Norm}(0), \text{Norm}(0)) &= \text{atan2}(\text{Norm}(0), \text{Norm}(0)) \\
\text{atan2}(\text{Norm}(c), \text{Norm}(a)) &= \text{atan2}(\text{Norm}(c), \text{Norm}(a)) \\
\text{atan2}(\text{Norm}(c), \text{Norm}(a)) &= \text{atan2}(\text{Norm}(c), \text{Norm}(a)) \\
\end{align*}
$$

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 $\text{abs}^#, \text{acos}^#, \text{asin}^#, \text{atan}^#, \text{atan2}^#, \text{ceil}^#, \text{cos}^#, \text{exp}^#, \text{floor}^#, \text{log}^#, \text{min}^#, \text{max}^#, \text{pow}^#, \text{random}^#, \text{round}^#, \text{sin}^#, \text{sqrt}^#, \text{tan}^#$ have been designed and implemented in the analyzer using the interval domain. We now give definitions for some of these abstract functions.

$$
\log^2(\text{Int}(a,b)) = \begin{cases} 
\mathbb{N}_N & \text{if } (b < 0) \\
\text{Int}(-\infty, -\infty) & \text{if } (a = 0 \land b = 0) \\
\text{Int}(0, 0) & \text{if } (a = 1 \land b = 1) \\
\text{Int}(+, +) & \text{if } (a = +\infty \land b = +\infty) \\
\text{Int}(\text{Log}(a), \text{Log}(b)) & \text{if } (a > 0) \\
\text{Int}(\text{Log}(a), \text{Log}(b)) & \text{if } (a \leq 0 \land b > 0) 
\end{cases}
$$

$$
\sqrt{\text{Int}(a,b)} = \begin{cases} 
\mathbb{N}_N & \text{if } (b < 0) \\
\text{Norm}(0, \sqrt{\text{Int}(a,b)}) & \text{if } (a \leq 0 \land b \geq 0) \\
\text{Int}(\text{sqrt}(a), \text{sqrt}(b)) & \text{otherwise} 
\end{cases}
$$

$$
\text{abs}^2(\text{Int}(a,b)) = \begin{cases} 
\text{Int}(a,b) & \text{if } (a \geq 0 \land b \geq 0) \\
\text{Int}(0, \text{max}(a, -b)) & \text{if } (a \leq 0 \land b \geq 0) \\
\text{Int}(0, -b-a) & \text{if } (a \leq 0 \land b \leq 0) 
\end{cases}
$$

The abstract mathematical functions were defined for all the elements of the extended interval domain. Above are the definitions for the $\log^2$, $\sqrt{\text{Int}(a,b)}$, $\text{abs}^2$ and $\text{atan2}^2$ on abstract elements $\text{Int}(a,b)$, $\text{Norm}(a,b)$, and $\text{atan2}(c)$ and $\text{Norm}(a,b)$. The function $\text{atan2}^2$ returns the arctangent of the quotient of its arguments which is a value between $-\pi$ and $\pi$. 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 $\text{assert}(i >= 0 \&\& i < F5.streams.length);$.

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 $\text{globalFuncList[TopNum]}$ with $\text{TopNum=Date.now()}$;
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 types of 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 TAJJS with the context sensitivity aspect which is entirely configurable by the user. SAFE is another static analyzer for JavaScript programs [16]. Unlike TAJJS 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 arrayIndexOutOfRange exceptions that could cause programs to crash or malfunction. 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


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.