IEEE754 Floating Point Bound Intervals for static analysis of JavaScript Programs

Astrid Younang and Lunjin Lu
Department of Computer Science and Engineering, Oakland University
Rochester Hills, Michigan, 48309, USA
(awaindja, L2Lu)@oakland.edu

Abstract

This paper extends our previous work on static analysis of JavaScript applications using an extended domain of intervals. The prior work uses integers as bounds of intervals, which leads to loss of precision. In this work, we use floating point numbers as bounds and design correct abstract operations on intervals that safely approximate concrete operations. We have also designed abstract functions that approximate concrete functions that manipulate numbers and return numbers as results. The experiments show an increase in precision on 5 out of 28 benchmark programs previously analyzed. The precision gain led to an average of 34% increase in analysis time.

keywords: JavaScript, static analysis, abstract interpretation, Floating Point Bounds

1 Introduction

JavaScript is a dynamic language that has gone from simple scripts in websites to large and complex programs for games, desktop and mobile applications. Compared to statically typed languages like C and Java, tools to aid developers in program understanding and debugging are still very few and at an early stage. In addition to that, achieving good precision is very difficult because JavaScript supports first class functions, dynamic property accesses and uses prototype-based inheritance. Because of this dynamism, statically estimating possible values of variables is difficult and often leads to results that are not precise. This imprecision in turn leads to many false positives.

This paper extends the results of our previous work on static analysis of JavaScript programs for type errors detection. The prior work used integers as bounds of intervals. However JavaScript programs contain IEEE754 compliant floating point operations such as addition, multiplication and division. Antoine Mine in [1] was the first to introduce floating point interval analysis. For each format (32 or 64 bit), he introduced the set of real intervals with floating point bounds. The default rounding mode of the IEEE754 standard is toward the nearest representable value. This means that a floating point number directly smaller or larger than the exact result can be chosen during rounding. IEEE754 provides 4 rounding modes: toward 0, toward the nearest representable value, toward $+\infty$ and toward $-\infty$. For our floating bound intervals, the lower bounds are rounded toward $-\infty$ and the upper bounds toward $+\infty$.

The framework used for our analysis is abstract interpretation, which is a theory of semantic approximations. Abstract interpretation enables the design of abstract domains for the analysis. Abstract domains can be view as sets of abstract elements along with abstract operations to manipulate them. This work focuses on abstract domains of numeric properties in general and interval domains in particular. Several abstract interpretation-based JavaScript static analyzers developed over the years [2], [3], [4] make use of the constant propagation domain for numbers as abstract numeric domain. The imprecision of this domain has led to many false positives. In our previous work [5], we proposed an extended domain of intervals to recover some lost precision. This work goes further by using floating point bound intervals for JavaScript static analysis.

The main contributions of this paper are as follows:

- An extended abstract domain of intervals with 64-bit IEEE754 floating point number bounds. This abstract domain is more precise than the ones used in JavaScript static analyzers such as TAJS [3], JSAI [6] and SAFE [4].

- An empirical performance and precision evaluation on benchmarks. The benchmarks used are Google V8 [7], SunSpider [8], browser addon programs from the Mozilla addon repository [9] and machine generated JavaScript code from the LLVM test suite [10]. The results show an increase in precision on 5 benchmark programs over a total of 28. The precision gain led to an increase in the analysis
performance.

2 Background - Floating Point representation

The IEEE standard was defined in response of the divergence of representations of real numbers on computers. Floating point numbers can be represented using 3 binary fields: a sign bit $s$, an exponent field $e$ and a fraction field $f$. The two main representations are the single precision(32-bit) and double precision(64-bit). The single precision numbers are represented with 8 bits for the exponent field and 23 bits for the fraction field for a total of 32 bits. The double precision numbers are represented with 11 bits for the exponent field and 52 bits for the fraction field for a total of 64 bits. There are two main standards: IEEE754 and IEEE854 and most computer manufacturers make use of the IEEE754 standard. The main operations specified by IEEE arithmetic are the addition, subtraction multiplication, division, square roots and remainders. The IEEE754 compliant floating point operations include the opposite of a floating point number can be exactly integers. The negation is an exact operation as the modulo, $\ll$, $\gg$, division. The rest of the operations that include the $\wedge$ return integers. The negation is an exact operation as the $\wedge$ and xor return integers. The negation is an exact operation as the $\wedge$ and xor return integers. The negation is an exact operation as the $\wedge$ and xor return integers. The negation is an exact operation as the $\wedge$ and xor return integers.

3 Floating Point Intervals

In this section, we present the floating point intervals, along with the approximation of some mathematical functions.

3.1 Extended domain of intervals

All JavaScript numbers are 64-bit floating point numbers as defined by the IEEE 754 standard. Let $\text{Float754}$ denote the set of all IEEE-754 numbers including the special numbers $\{\text{NaN}, +\infty, -\infty\}$. The extended abstract domain of intervals is defined as follows:

\[ I^2 = \{\top, \bot, \text{NaN, Int32}\} \cup \{\text{Int}(a, b), \text{Norm}(a, b)\mid (a, b \in \text{Float754}) \land a \leq b\} \]

\text{Norm}(a, b) describes the set of real numbers between $a$ and $b$ including $\text{NaN}, \text{Int}(a, b)$ the same set of real numbers without $\text{NaN}$. Int32 the set of all unsigned 32-bit integers. Let $U32\text{Max}$ be the maximal integer of 32 bits - 4294967295.

The partial order $\sqsubseteq$ over $I^2$ is formally defined as follows:

\[
\begin{align*}
\bot & \sqsubseteq i & \text{for any } i \in I^2 \\
\text{NaN} & \sqsubseteq \text{NaN} \\
\text{NaN} & \sqsubseteq \text{Norm}(a, b) \\
\text{Int32} & \sqsubseteq \text{Int}(a, b) & \text{iff } a \leq 0 \land U32\text{Max} \leq b \\
\text{Int32} & \sqsubseteq \text{Norm}(a, b) & \text{iff } a \leq 0 \land U32\text{Max} \leq b \\
\text{Int}(a, b) & \sqsubseteq \text{Int}(c, d) & \text{iff } a \geq c \land b \leq d \\
\text{Int}(a, b) & \sqsubseteq \text{Norm}(c, d) & \text{iff } a \geq c \land b \leq d \\
\text{Norm}(a, b) & \sqsubseteq \text{Norm}(c, d) & \text{iff } a \geq c \land b \leq d \\
\end{align*}
\]

Let $\text{Val}$ be the set of concrete JavaScript numeric values. $\text{Val} = \text{Float754}$. The concretization function $\gamma : I^2 \longrightarrow P(\text{Val})$ is defined as follows:

\[
\begin{align*}
\gamma(\bot) &= \{\} \\
\gamma(\text{NaN}) &= \{\text{NaN}\} \\
\gamma(\text{Norm}(a, b)) &= \{r\mid r \in \text{Float754} \setminus \{\text{NaN}\}, a \leq r \leq b\} \\
\gamma(\text{Int}(a, b)) &= \{r\mid r \in \text{Int32} \setminus \{\text{NaN}\}, a \leq r \leq b\} \\
\gamma(\text{Int32}) &= \{r\mid r \in \mathbb{Z}, 0 \leq r \leq 2^{32} - 1\} \\
\gamma(\top) &= \text{Val}
\end{align*}
\]

3.2 Abstract operations

The IEEE754 compliant floating point operations include the addition, subtraction, multiplication and division. The rest of the operations that include the modulo, $\ll$, $\gg$, $\wedge$, bitwise and, or and xor return integers. The negation is an exact operation as the opposite of a floating point number can be exactly represented.

Operations such as $\infty \circ 0$ or $0 \circ 0$ result in the special number $\text{NaN}$. Let an interval $i \in I^2$. Let functions $f_0, f_1 : I^2 \longrightarrow \mathbb{B}$ be defined such that $f_0(i)$ is true if and only if $0 \in \gamma(i)$ and $f_1(i)$ is true if and only if $\gamma(i)$ contains $\infty$. The functions $f_0$ and $f_1$ are used by multiplicative operators $\odot$, $\circ$ and $\text{mod}$ to check whether or not an interval contains 0 or $\infty$. This is to track possible operations that might result to $\text{NaN}$ and adjust the resulting intervals accordingly.

3.3 IEEE754 Floating Point bounds

When a number cannot be exactly represented, the rounding mode indicates how it is treated. As example, the $\pi$ value is respectively $3.1415925025939941406250$ and $3.141592741012573241875$ when rounded toward $-\infty$ and $+\infty$ respectively. The C function
fesetround(int rdir) sets the rounding as specified by its arguments and returns 0 when the rounding is successful and non-zero when an error occurs during rounding.

The fesetround(int rdir) function is used in our work to round our interval bounds. The lower and upper bounds are all rounded toward $+\infty$. To round the lower bounds back to $-\infty$, they are negated during an arithmetic operation as the opposite of a floating point number can be exactly represented. Let $\text{Int}(a, b)$, $\text{Int}(c, d) \in I^2$, For the addition operation, we have: $\text{Int}(a, b) + \text{Int}(c, d) = \text{Int}(-((a) + (-c)), b + d)$

More details on the corner cases of the main JavaScript operations can be found in [5].

### 3.4 Approximation of math functions

Abstract mathematical functions that approximate concrete functions were designed using the floating point bound interval domain. Let $i, i_1, i_2, \text{Int}(a, b)$, $\text{Int}(c, d)$, $\text{Norm}(a, b)$, $\text{Norm}(c, d) \in I^2$, $a, b, c, d \in \text{Float}754$, the function $\text{acos}(.)$ can be formally designed as follows:

<table>
<thead>
<tr>
<th>$\text{Int}(a, b)$</th>
<th>$\text{Int}(\text{acos}(a), \text{acos}(b))$ if $a = b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{Norm}(a, b)$</td>
<td>$\text{Norm}(\text{acos}(a), \text{acos}(b))$ if $a = b$</td>
</tr>
</tbody>
</table>

The design of the mathematical functions $\text{abs}(.)$, $\text{asin}(.)$, $\text{atan}(.)$, $\text{atan2}(.)$, $\text{ceil}(.)$, $\text{cos}(.)$, $\text{exp}(.)$, $\text{floor}(.)$, $\text{log}(.)$, $\text{pow}(.)$, $\text{random}(.)$, $\text{round}(.)$, $\text{sin}(.)$, $\text{sqrt}(.)$, $\text{tan}(.)$ follows the same reasoning and their entire specifications were omitted for space reasons.

## 4 Experiments and Evaluation

This section describes in more details the use of the C function fesetround(int rdir) and the evaluation results.

### 4.1 Experiments

We use the abstract interpreter JSI to conduct our experiments. JSI is a JavaScript Static Analyzer written in Scala. To call the C function in Scala, we use the SWIG (Simplified Wrapper and Interface Generator) framework. SWIG [11] is a software tool that generates bindings between C/C++ code and scripting languages like Tcl, Perl, Python, Java and C#.

Figure 1 illustrates an overview of the Scala-C interface. SWIG takes as input a C file with the needed C functions and an interface file. The interface file contains the functions and variable declarations. Depending on the target language, SWIG will generate wrapper classes for the C functions and store them in a shared library, which will be used to call the needed C functions.

![Figure 1: Overview of the Scala-C interface. The SWIG framework takes a C/C++ and an interface files as inputs and generates wrappers for a particular language (Python, Java or Tcl). Those wrappers are put in a library than can be used to call the C functions.](image)

The interface file contains C function prototypes and variable declarations.

### 4.2 Evaluation

We ran JSI on a Scientific Linux 6.3 distribution with 24 Intel Xeon CPUs with a capacity of 1.6GHz and 32GB memory. The benchmarks used are SunSpider [8], Google V8 [7], browser addon programs from the Mozilla addon repository [9] and machine generated JavaScript code from Emscripten LLVM test suite [10].

#### 4.2.1 Precision:

The precision metric used is the number of program locations that may generate a type error warning. A type error warning occurs in situations where a non-function value is called as a function, a property of a null/undefined object is read, updated or deleted. The early detection of possible type errors in JavaScript
programs is as important as type errors are not silent and cause programs to crash or malfunction. Figure 2 illustrates the number of type errors warnings reported by the 3 different analyses. The first analysis is the original analysis with the constant propagation domain for numbers as abstract numeric domain. The second analysis integrates the interval domain as abstract domain. At the end of the second analysis, we noticed a reduction in the number of type error warnings reported. The third analysis is ran with IEEE 754 64-bit floating point bounds for the intervals. The gain in precision, although small, is noticed in 5 benchmark programs. The number of type error warnings reported in **adn-coffee_pods_deals**, **std-access-nbody**, **std-string-base64** and **std-date-format-tofte** are 5-4-4, 6-6-6, 3-2-2, and 3-2-2 for the first, second and third analysis respectively. They cannot be seen on figure 2 because of the format of the data labels.

4.2.2 Performance:

Figure 3 illustrates the percentage increase of the running time of the third analysis compared to the second analysis. The average increase is 34%. This increase can be explained by the C function call through the SWIG generated wrappers.

5 Related Work

In this section, we will present the major abstract numeric domains, abstract interpretation-based static analyzers and their numeric domains.

5.1 Abstract numeric domains

Verifying the correctness or improving the performance of certain applications sometimes requires reasoning about numeric properties. Over the years, several numeric domains have been proposed and they can be divided into two main categories: non-relational and relational.

In non-relational abstract numeric domains, the relationship between variables is lost during abstraction. Non-relational numeric abstract domains include sign, constant propagation, interval and simple congruence domains. For the sign domain, only the sign of the numeric variable is kept during abstraction. The constant propagation domain determines when a variable takes constant values throughout the execution of the program. The interval domain for program analysis, introduced by Cousot in [12] enables the value of variable to be abstracted by an interval where the lower and upper bounds respectively represent the lowest and greatest value that the variable can take. The integer congruence domain, introduced by Granger et al. in 1979 discovers properties like \textit{the variable }$x$\textit{ is congruence to }$x \mod m$\textit{ where }$x$\textit{ and }$m$\textit{ are automatically...}
In relational domains, the relationship between variables is kept during the abstraction. The most popular relational abstract domains include the polyhedral and octagon domains. The polyhedra domain was introduced by Patrick Cousot and Nicolas Halbwachs in [13]. A polyhedron is of the form $Ax \leq b$ where $A$ is matrix of the dimension of the variables in the programs and $x$ and $b$ vectors. The octagon domain is a weakly relational domain introduced by Antoine Mine in [14]. The domain is based on Difference-Bound Matrices and represents invariants of the form $(\pm x \pm y \leq c)$ where $x$ and $y$ are variables and $c$ a constant.

5.2 JavaScript static analyzers and their abstract numeric domains

TAJS (Type Analysis for JavaScript) is a static analyzer developed by Jensen et al [3] to tackle the dynamic typing of JavaScript. It provides sound and detailed type in JavaScript programs and detects type related errors. Their abstract numeric domain is the constant propagation domain for numbers extended with three additional elements - INF, UInt and NotUInt. INF is the least upper bound of $-\infty$ and $+\infty$, UInt that of all unsigned integers and NotUInt that of all other float-point numbers.

JSAI (JavaScript Abstract Interpreter) is a JavaScript analysis platform developed by Kashyap et al in [2]. It has been used for many client analyses such as type and range errors detection [6], program slicing [15] and security auditing of browser addons [16]. Their abstract numeric domain is the constant propagation domain augmented with an abstract element called $NReal$, which is the set of all unsigned 32-bit integers.

SAFE (Scalable Framework for EcmaScript) is a promising static analysis framework for JavaScript [4]. It provides 3 intermediate representations for JavaScript and can be used for many client analyses. Compared to TAJS and JSAI, SAFE does not use the Rhino [17] parser which uses EcmaScript 3 and supports EcmaScript 5. SAFE uses the same abstract numeric domain as TAJS.

RATA (Rapid Atomic Type Analysis) is a JavaScript analyzer used by a Microsoft JavaScript engine [18]. Compared to TAJS, JSAI and SAFE which mainly support bug detection, RATA is meant for JIT compiler optimization. It infers type information that is used by the JavaScript engine to allocate specific registers for integers or floating point numbers. In RATA, the abstract numeric domain contains abstract elements NaN, Normal$(a,b)$, OpenLeft$(a)$, OpenRight$(a)$. Normal$(a,b)$ is the set of real numbers between $a$ and $b$ with $a,b \in \mathbb{Z}$, including NaN. OpenLeft$(a)$ and OpenRight$(a)$ are the sets of real numbers that are less than $a$ and greater than $a$ respectively with $a \in \mathbb{Z}$, including NaN. All abstract elements in RATA’s interval domain contain NaN, our interval domain has abstract elements that do not contain NaN such as Int$(\cdot, \cdot)$ and Int32. Also, intervals in RATA have integer bounds.
6 Conclusion and Future Work

We presented an IEEE754 floating point bound interval domain for JavaScript program analysis. We designed and implemented abstract operations and abstract mathematical functions over this domain. The precision gain in terms of type error warnings was noticed in 5 over 28 benchmark programs previously analyzed. This gain led to a 34% increase in running time. Future work will investigate the trade-off between precision and cost of the use of relational numeric domains for static analysis of JavaScript programs.

References


