EPiC Series in Computing

Volume 58, 2019, Pages 155–167

Proceedings of 34th International Conference on Computers and Their Applications

Analyzing JavaScript Programs Using Octagon Domain

Nabil Almashfi and Lunjin Lu

Oakland University, Rochester, Michigan, USA
{nalmashfi,L2Lu}@oakland.edu

Abstract

Static analyzers for JavaScript use constant propagation and interval domains to discover numerical properties of program variables. These domains are non-relational and incapable of tracking relationships between variables, leading to imprecise analysis. This paper presents a static analyzer for the full language of JavaScript that employs the octagon domain to capture numerical properties of the program. Our work is built on top of TAJS (type analyzer for JavaScript) which employs a constant propagation domain for numerical properties. We reengineered TAJS's abstract domain for abstractions of primitive values and its abstract domain for object abstractions and related transfer functions, resulting in an analyzer that is much more precise. Our experiments show an improvement in analysis precision of JavaScript programs with an acceptable increase in cost.

1 Introduction

JavaScript has a very broad presence in web applications. It has become one of the most popular programming languages for Web applications. However, JavaScript allows for the creation of numerous bugs during development because of its flexibility. This flexibility may produce subtle programming errors that are not reported by the language system.

Static program analysis is promising in uncovering subtle program bugs. It examines programs and discovers their properties that hold for all possible execution paths of the program. There are various approaches to static analysis. The static analysis presented in this paper is based on abstract interpretation [3]. In abstract interpretation, a program is analyzed to model some properties of concrete computations which gives an understanding of the program's behavior.

Numeric analyses have been a topic of research over many years. Many numeric domains have been proposed varying in precision and efficiency. The existing numeric domains can be divided into two types: non-relational domains and relational domains. Non-relational domains collect simple properties of variables and variables are abstracted independently of each other. On the other hand, relational domains collect properties of variables and take the relationships between variables into consideration.

The octagon abstract domain [13] is a relational domain that enables the discovery of relationships between program variables. In the case of our work, it allows us to discover the relationships among numeric variables, numeric elements of arrays and numeric properties of objects.

In this paper, we present a static analyzer that supports relational abstractions. Specifically, the analyzer uses the octagon domain to track relational numeric properties of program variables. The analyzer, which we will refer to as TAJS_{oct}, is built on top of TAJS [6]. TAJS tracks numeric properties of different program variables separately and is strictly non-relational. The reengineering of TAJS to support relational abstractions is a non-trivial task that requires the reengineering of TAJS's abstract domain for primitive value abstractions, its abstract domain for object abstractions and related abstract operations.

We have validated the precision and performance of TAJS_{oct} using a suite of JavaScript benchmark programs. Our experiments suggest that TAJS_{oct} infers more precise numerical properties and that constant propagation domains are very imprecise and yield a high number of false positives that can be avoided using relational domains.

The main contributions of this paper are as follows. Firstly, we discuss state-of-the-art JavaScript analyzers and their numerical domains. Secondly, we describe the implementation of TAJS_{oct} that uses the octagon domain to track relational numeric properties of program variables. Finally, we present an empirical evaluation of performance and precision of the analyzer on different JavaScript programs.

The rest of the paper is structured as follows. Section 2 presents related work. Section 3 provides the motivation behind this work with a motivating example. Section 4 gives an introduction to TAJS. Section 5 describes the abstract domains in TAJS_{oct}. Section 6 describes some of the transfer functions. Section 7 presents the evaluation results in terms of precision and performance and section 8 concludes.

2 Related Work

Several static analyzers based on abstract interpretation have been developed for JavaScript. In this section, we focus on those analyzers that capture numerical properties of program variables. Most of these tools use constant propagation domains to infer type information about variables.

TAJS [6], Type Analysis for JavaScript, is a dataflow analysis tool based on abstract interpretation. It can infer type information and call graphs. The numerical abstract domain used in TAJS is a constant propagation domain for numbers augmented with four abstract elements UInt, NotUInt, Inf and NaN. Inf represents $\pm\infty$, NaN represents "Not-a-Number" value, UInt represents 32-bit signed integers and NotUInt is for all other numbers.

JSAI [8], JavaScript Abstract Interpreter, is also an abstract interpreter for JavaScript programs. It detects and reports type errors in JavaScript programs and it has a range of context-sensitivities from which a user can choose. The numerical and string abstract domains in JSAI are configurable. However, the default numerical domain is a constant propagation domain augmented with abstract elements similar to those in TAJS except that Inf is used to represent $+\infty$ and NInf is used for $-\infty$.

SAFE [9], a Scalable Analysis Framework for EcmaScript, is another JavaScript analyzer. It detects type-related errors as well as reference errors, dead code, null/undefined variables and more. The numerical abstract domain is the same as JSAI's except that the abstract elements have slightly different names.

RATA [11], Rapid Atomic Type Analysis, is a combination of interval analysis, kind analysis and variation analysis. The kind analysis enables RATA to determine whether the value of a variable is definitely a 32-bit integer. The variation analysis relates the values of variables on a per-loop basis. It over-approximates for each variable in a single loop its values by an interval. This information helps Just-in-time (JIT) compilers to generate more specialized machine instructions which results in large performance gains.

```
1   var i = 10, a = 1;
2   while (i > 0) {
3       a = a + 1;
4       i = i - 1;
5   }
6   if (a < 10) { ... }
7   else { ... }
```

$O_1 = \top$
$O_2 = \{i = 10\}$
$O_3 = \{i = 10, a = 1, i + a = 11, i - a = 9\}$
$O_{4,1} = \{i = 10, a = 1, i + a = 11, i - a = 9\}$
$O_{5,1} = \{i = 10, a = 2, i + a = 12, i - a = 8\}$
$O_{6,1} = \{i = 9, a = 2, i + a = 11, i - a = 7\}$
$O_{4,2} = O_{4,1} \bigtriangledown (O_{6,1})_{(i>0)}$
$O_{4,2} = \{1 \leq i \leq 10, 1 \leq a \leq 10, i + a = 11, -9 \leq i - a \leq 9\}$
$O_{5,2} = \{1 \leq i \leq 10, 2 \leq a \leq 11, i + a = 12, -10 \leq i - a \leq 8\}$
$O_{6,2} = \{0 \leq i \leq 9, 2 \leq a \leq 11, i + a = 11, -11 \leq i - a \leq 7\}$
$O_{4,3} = O_{4,2}$ (fixpoint reached)
$O_7 = \{i = 0, a = 11, i + a = 11, i - a = -11\}$

Figure 1: Simple JavaScript program

All JavaScript Analyzers presented use non-relational numeric abstract domains and therefore are unable to capture relational information between numerical program variables.

3 Motivation

In this section, we use a simple JavaScript program to demonstrate the impact of our analysis on precision. Fig. 1 shows the need for a relational domain. In this example, we have a while loop with a loop counter i that is decremented at every iteration. On the other hand, the variable a is incremented by one at every iteration. The variable a is then used in a conditional statement where the condition is true if the value of a is less than 10, false otherwise. Fig. 1 also shows the computation of the octagonal constraints that result from the analysis of the program. $O_{\text{i,j}}$ represents the octagonal constraints before the execution of line i where j is the number of iteration inside the loop.

At the beginning of the analysis, the initial octagon is set to the top element, $O_1 = \top$, and information is propagated through the control flow of the program execution. Widening $\bigtriangledown$ operator is used to ensure the analysis reaches a fixpoint. Our analysis of JavaScript programs using the octagon domain can discover the relational loop invariant $i + a = 11$ at line 2. Using widening operator, the analysis can also infer that $i = 0$ and conclude that $a = 11$ at line 6. Given this information, it is clear that the condition at line 6 will always evaluate to false and the else branch will be marked as dead code. The constant propagation domain, used in TAJS, and the interval numerical domain, which has been applied to JavaScript, are non-relational domains and therefore not able to discover this information.

4 Static Analyzer TAJS

TAJS [6] is a flow-sensitive analyzer for JavaScript. It can infer type information for JavaScript programs using abstract interpretation and the monotone framework. We recall from [6] the analysis lattice:

$$V \times N \to \mathsf{State}$$

where V denotes contexts and N is the set of program points. State consists of an abstract *store* and an abstract *stack*. The abstract *stack* models the JavaScript runtime stack. The *store* is used for the abstraction of values and it is defined as a partial map from *object labels* L to abstract *objects* where *object labels* are possible allocation sites.

$$\mathsf{Store} = (L \rightharpoonup \mathsf{Object})$$

Abstract *objects* are maps from property names P to values and flags as follows:

$$\mathsf{Object} = \quad (P \to \mathsf{Value} \times \mathsf{Absent} \times \mathsf{Attributes} \times \mathsf{Modified}) \times \wp(\mathsf{ScopeChain})$$

where Absent models potentially absent properties, $\mathsf{Modified}$ models modified properties in interprocedural analysis, $\mathsf{Attributes}$ models the property attributes $\mathsf{ReadOnly}$, $\mathsf{DontDelete}$, and $\mathsf{DontEnum}$. $\mathsf{ScopeChain}$ is used with objects representing functions to track execution contexts. The Value component is used to model all possible values using a non-relational abstraction.

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

Undef and Null represent undefined and null values respectively. Bool, Num and String are boolean, number and string domains respectively.

5 Abstract Domains

TAJS$_{\text{oct}}$ preserves the flow sensitivity of TAJS. In order to cope well with a relational abstraction for numerical variables, we use two abstract domains for numerical abstractions. The first domain is the octagon abstract domain which is used to track relationships between variables. The second domain is the type abstract domain which is used to determine whether the value of a variable is an unsigned integer number or a float number. We describe both domains next.

5.1 Octagon Abstract Domain

The octagon abstract domain [13] was used in order to precisely track values of numeric variables and the relationships between them. We recall that the octagon abstract domain captures relations in the form of $(\pm x \pm y \leq c)$, where x and y are variables, $\pm \in \{-, 0, +\}$ and c is a constant. Any constraints of the form $(\pm x \pm y \leq c)$ are called octagonal constraints or octagonal inequalities. An octagon can be defined as the conjunction of a set of octagonal inequalities. Each octagonal inequality can be represented as an element of a difference-bound matrix. The complete lattice of octagons is $(O_+, \sqsubseteq_o, \sqcup_o, \sqcap_o, \perp_o, \top_o)$ where $O_+ = O \cup \{\perp_o\}$ and O is the set of octagons. The partial order between two octagons $a \sqsubseteq_o b$ means that each inequality in a is as tight as or tighter than the corresponding inequality in b. $\sqcup_o$ and $\sqcap_o$ represent the least upper bound and greatest lower bound respectively. $\perp_o$ represents the bottom element and $\top_o$ represent the top element.

In our analysis, each variable is mapped to an allocation site and is represented by a dimension in the octagon. However, many variables could be mapped to one allocation site. The global object label, for instance, is an allocation site to which variables declared in the global scope are mapped.

Since JavaScript variables can be of any type, we do not have dimensions for variables that do not hold numerical values. Assigning a numerical value to a variable cause a new dimension to be added representing the value of that variable. On the other hand, assigning a non-numerical value to a variable holding a numerical value causes the corresponding dimension to be deleted. Object properties are also treated in the same way. As a consequence of this approach, the size of octagons may dynamically change during the analysis and we follow [10] to call these octagons dynamic. Each dimension in a dynamic octagon is a pair of a variable name v and an allocation site l. The allocation site l serves as the abstract address to the variable v. We define the abstract domain of dynamic octagons next.

Definition 1: *(Dynamic Octagon). Let L be the set of possible allocation sites and V the set of variables. We can define the abstract domain of dynamic octagons as*

$$DynamicOctagon = (\bigcup_{P \subseteq L \times V} O_P, \sqsubseteq_d, \sqcup_d, \sqcap_d, \bot_d, \top_d)$$

Adding and removing dimensions from dynamic octagons complicates the analysis. For instance, the standard join operation $\sqcup_o$ on two octagons o_1 and o_2 assumes that both o_1 and o_2 have the same size and represent the same variables which is not the case in our analysis. As a consequence, the partial order $\sqsubseteq_d$ as well as standard abstract operations on octagons need to be redefined.

Intuitively, the partial order between two dynamic octagons $d_{p_1} \sqsubseteq_d d_{p_2}$ means that each constraint in d_{p_1} is as tight as or tighter than the corresponding constraint in d_{p_2}. However, d_{p_1} may contain constraints that have no corresponding constraints in d_{p_2} which makes d_{p_2} an over-approximation of d_{p_1}. Therefore, we require that $p_2 \subseteq p_1$. Formally,

$$d_{p_1} \sqsubseteq_d d_{p_2} \quad \textbf{iff} \quad p_2 \subseteq p_1 \textbf{ and } \pi_{p_2}(d_{p_1}) \sqsubseteq_o \pi_{p_2}(d_{p_2})$$

where $\pi_p(d)$ is the projection of the dynamic octagon d on the pairs of variables and allocation sites p.

Example: Let $d_{p_1} = \{(l_1.v_1 \leq 15) \wedge (l_1.v_1 + l_1.v_2 \leq 20)\}$, $d_{p_2} = \{(l_1.v_1 \leq 20)\}$. Then $d_{p_1} \sqsubseteq_d d_{p_2}$.

We recall from [10] the abstract operations join $\sqcup_d$, meet $\sqcap_d$ and widening ∇_d on dynamic octagons with some changes to ensure the correctness of the analysis. We observed that the definitions of these abstract operations in [10] do not always yield a correct analysis. Particularly, when an object containing an integer field is allocated memory in one of the branches of a conditional, the analysis treats the value of the integer field in the other branch as $\bot$, when it is in fact not defined. Therefore, when two dynamic octagons are joined, the constraints that are present in one octagon but not the other are simply added to the resulting dynamic octagon which yields incorrect analysis.

We redefine the abstract operations join $\sqcup_d$, meet $\sqcap_d$ and widening ∇_d on dynamic octagons and we use $\sqcup_o$, $\sqcap_o$ and ∇_o for the standard join, meet and widening operators respectively.

Given two dynamic octagons d_{p_1} and d_{p_2}, the kernel of the d_{p_1} and d_{p_2} is defined as $p_1 \cap p_2$. The join $\sqcup_d$, meet $\sqcap_d$ and widening ∇_d operators are defined as follows.

Join. There are two cases to consider when joining two dynamic octagons d_{p_1} and d_{p_2}. If $p_1 = p_2$, we simply apply the standard join operation on d_{p_1} and d_{p_2}. If $p_1 \neq p_2$, we construct a dynamic octagon by using the standard join $\sqcup_o$ on the constraints involving the variables in the kernel of d_{p_1} and d_{p_2} and we discard all the remaining constraints. The join of dynamic octagons $d_{p_1} \sqcup_d d_{p_2}$ is defined by Eq.(1) in Fig. 2.

$$
\begin{array}{rcll}
p & = & p_1 \cap p_2 & \\
\kappa_1 & = & \pi_p(d_{p_1}) & \\
\kappa_2 & = & \pi_p(d_{p_2}) & \\
\eta_1 & = & \pi_{p_1-p}(d_{p_1}) & \\
\eta_2 & = & \pi_{p_2-p}(d_{p_2}) & \\
d_{p_1} \sqcup_d d_{p_2} & = & (\kappa_1 \sqcup_o \kappa_2) & (1)\\
d_{p_1} \sqcap_d d_{p_2} & = & (\kappa_1 \sqcap_o \kappa_2) \cup \eta_1 \cup \eta_2 & (2)\\
d_{p_1} \nabla_d d_{p_2} & = & (\kappa_1 \nabla_o \kappa_2) & (3)
\end{array}
$$

Figure 2: The definitions of join, meet and widening operations $\sqcup_d$, $\sqcap_d$ and ∇_d on dynamic octagons where $\pi_p(d)$ is the projection of the dynamic octagon d on p.

Example 1: Let $p_1 = \{(l_1, v_1), (l_1, v_2)\}$, $d_{p_1} = \{(l_1.v_1 \leq 15) \wedge (l_1.v_1 + l_1.v_2 \leq 20)\}$, $p_2 = \{(l_1, v_1), (l_1, v_2), (l_2, v_3)\}$, $d_{p_2} = \{(l_1.v_1 \leq 10) \wedge (l_1.v_1 + l_1.v_2 \leq 25) \wedge (l_2.v_3 \leq 5)\}$. The kernel of d_{p_1} and d_{p_2} is $\{(l_1, v_1), (l_1, v_2)\}$ and we have the following:

$$
\begin{array}{rcl}
\kappa_1 & = & \{(l_1.v_1 \leq 15) \wedge (l_1.v_1 + l_1.v_2 \leq 20)\}\\
\kappa_2 & = & \{(l_1.v_1 \leq 10) \wedge (l_1.v_1 + l_1.v_2 \leq 25)\}\\
d_{p_1} \sqcup_d d_{p_2} & = & \kappa_1 \sqcup_o \kappa_2\\
d_{p_1} \sqcup_d d_{p_2} & = & \{(l_1.v_1 \leq 15) \wedge (l_1.v_1 + l_1.v_2 \leq 25)\}
\end{array}
$$

Meet. There are two cases to consider when meeting two dynamic octagons d_{p_1} and d_{p_2}. If $p_1 = p_2$, we simply apply the standard meet operation on d_{p_1} and d_{p_2}. If $p_1 \neq p_2$, we first construct a dynamic octagon by using the standard meet $\sqcap_o$ on the constraints involving the variables in the kernel of d_{p_1} and d_{p_2}. Then we add all the remaining constraints to the resulting dynamic octagon. The meet of dynamic octagons $d_{p_1} \sqcap_d d_{p_2}$ is defined by Eq.(2) in Fig. 2.

Example 2: Let d_{p_1}, d_{p_2} be defined as in Example 1. The we have the following:

$$
\begin{array}{rcl}
\kappa_1 \sqcap_o \kappa_2 & = & \{(l_1.v_1 \leq 10) \wedge (l_1.v_1 + l_1.v_2 \leq 20)\}\\
d_{p_1} \sqcap_d d_{p_2} & = & \{(l_1.v_1 \leq 10) \wedge (l_1.v_1 + l_1.v_2 \leq 20)\\
& & \wedge (l_2.v_3 \leq 5)\}
\end{array}
$$

Widening. The widening of two dynamic octagons is similar to the join operation in that we construct a dynamic octagon by using the standard widening ∇_o on the constraints involving the variables in the kernel of d_{p_1} and d_{p_2} and we discard all the remaining constraints. The widening of dynamic octagons $d_{p_1} \nabla_d d_{p_2}$ is defined by Eq.(3) in Fig. 2.

Floating-Point Octagons. Unlike many other programming languages, all numbers in JavaScript are floating-point. They are always stored as 64-bit floating-point based on the international IEEE 754 standard. As a consequence, we adopt the framework presented in [12] in order to soundly abstract floating-point computations in JavaScript. Floating-point expressions are expressed using interval linear forms on real numbers that can be fed to octagons and rounding errors are abstracted as non-deterministic error intervals.

5.2 Number Type Domain

The number type domain is used to determine whether the value of a variable is an unsigned integer number or a float number. It also determines the possibility that a variable could be

NaN indicating that a number is not a legal number. This domain does not contain an abstract element that represents infinity since the infinity value can be captured by the octagon domain. The motivation behind this domain is to increase analysis precision when analyzing arrays. The set of elements of the type abstract domain is:

$$\mathsf{Type}^\sharp = \{\bot_t\,, \mathsf{UInt}\ , \mathsf{Float}\ , \mathsf{NaN}\ ,\ \top_t\}$$

The abstract element UInt represents unsigned integers and Float represents all other numbers. Let NumVal be the set of concrete numerical values $\mathsf{NumVal} = \mathbb{R} \cup \{\mathsf{NaN}\}$ where $\mathbb{R}$ is the set of real numbers. The concretization function $\gamma_t : \mathsf{Type}^\sharp \to \wp(\mathsf{NumVal})$ is defined as follows:

$$\begin{array}{rcl} \gamma_t(\bot_t) & = & \emptyset \\ \gamma_t(\mathsf{NaN}) & = & \{\mathsf{NaN}\} \\ \gamma_t(\mathsf{UInt}) & = & \{\, r \mid r \in \mathbb{Z}, 0 \leq r \leq 2^{31} - 1\,\} \cup \{\mathsf{NaN}\} \\ \gamma_t(\mathsf{Float}) & = & \{\, r \mid r \in \mathbb{R}, r\ \mathit{is}\ \mathsf{Float754}\} \cup \{\mathsf{NaN}\} \\ \gamma_t(\top_t) & = & \mathsf{NumVal} \end{array}$$

where Float754 is the set of all 64-bits IEEE754 numbers. The abstraction function α_t : $\wp(\mathsf{NumVal}) \to \mathsf{Type}^\sharp$ is defined as

$$\alpha_t(S) = \sqcup_t\{\, \hat{\alpha_t}(n) \mid n \in S\} \text{ where}$$

$$\hat{\alpha_t}(n) = \begin{cases} \mathsf{NaN} & \text{if n is NaN} \\ \mathsf{UInt} & \text{if n is an unsigned integer} \\ \mathsf{Float} & \text{if n is a floating-point number} \\ \top_t & \text{otherwise} \end{cases}$$

Theorem 1: (α_t, γ_t) is a Galois connection.

5.3 Abstract String Domain

The string domain interacts closely with the number domain when analyzing arrays since array indices are represented with numeric string properties as well as non-numeric string properties. The original string domain in TAJS is equipped with the two abstract elements UIntString and NotUIntString in order to increase precision when dealing with arrays. UIntString describes the set of string representations of unsigned integers whereas NotUIntString describes the set of string representations of all other numbers and strings. However, this domain is still inadequate since it could possibly affect unrelated properties that fall under the category UIntString.

The projection operator in the octagon domain enables us to have an interval representing the values that a variable might hold. Therefore, we exploit the increase in precision in the number domain to develop a string domain that can capture precise information on array index ranges. This domain is a part of a static conventionality analysis for JavaScript arrays that was introduced in [15] which gives more precise information about array index ranges. The abstract string domain is a constant propagation domain similar to the string domain in TAJS except that it is extended with integer intervals as follows.

$$\begin{aligned}\mathsf{String}^\sharp \quad = \quad & \{\bot_s,\top_s,\mathsf{UIntString},\mathsf{NotUIntString}\} \cup \{[a,b] \mid a,b \in \mathbb{N} \wedge 0 \le a \le b \le IdMax\} \\ & \cup\ \mathsf{String}\ \setminus \{toString(i) \mid 0 \le i \le IdMax, i \in \mathbb{N}\}\end{aligned}$$

where String represents the set of all strings and $toString()$ is a function that maps a number to its string representation. In JavaScript, an array index is a property whose name is a string representing an integer between 0 and IdMax ($2^{32} - 2$).

The number type domain and string domain are integrated in the Value component described previously.

$$\mathsf{Value} = \mathsf{Undef} \times \mathsf{Null} \times \mathsf{Bool} \times \mathsf{Type}^\sharp \times \mathsf{String}^\sharp \times \wp(L)$$

5.4 Abstract Store

The abstract store in TAJS uses non-relational domains to abstract values. The use of a relational domain to abstract numbers requires the reengineering of the store. As a consequence, we redefine the abstract store as the follows:

$$\mathsf{Store} = (L \rightarrow Object) \times \mathsf{DynamicOctagon}$$

where the first element of the cartesian product is non-relational abstraction for all values except numbers and DynamicOctagon is for relational abstraction for numbers.

6 Transfer Functions

We describe in this section some of the transfer functions and our approach to handle the complications of JavaScript in order to increase precision, yet keep good performance.

6.1 Assignments

JavaScript is a dynamically typed language and a given variable can hold values of different types: numbers, strings, objects and more. The addition operator can also be used for concatenation as well as adding numbers. As a consequence, we first need to resolve the types of the operands and determine the correct behavior before we can operate on variables. However, numerical assignments, which we are interested in, can be divided into two kinds: one kind is where assignments can be exactly modeled in the octagon domain and the other kind is where assignments are approximated as described in [13]. Assigning a numerical value to a variable causes a new dimension to be added in the octagon to represent that value. On the other hand, changing the value of a variable from a numerical value to a non-numerical value causes the corresponding dimension to be deleted. For instance, if we have the following

```
var x = "s1";
x = 10;
x = "s2"
```

The variable x will not be mapped to a dimension in the octagon after analyzing the first line. Analyzing the second line will cause a new dimension to be created to represent the numerical value of x. Assigning a string to x in the third line causes the corresponding dimension to be deleted since x no longer holds a numerical value.

6.2 Function calls

When a function in JavaScript is called, the arguments may be either passed by value or by reference and we distinguish between these two cases:
- If the arguments are passed by value, we add a new dimension in the octagon to represent each argument as long as the arguments hold numerical values.
- If the arguments are passed by reference, which occurs in the case of passing an instance of an object as an argument, there will be no new dimensions added to the octagon. Any changes made to properties of the object within the function will be reflected on the original properties. For instance, if we have the following

```
var p1 = {id: 20};
var x = 10;
fun1(x, p1);
function fun1(y, p2) {
    p2.id = 10;
}
```

When analyzing this program, the octagon will contain the following dimensions $\{l_1.id, l_2.x, l_3.y\}$. The two variables $p1, p2$ refer to the same allocation site l_1 and therefore they correspond to the same dimension in the octagon when updating the property id which is $l_1.id$.

6.3 Arrays

Arrays in JavaScript are different from other programming languages. For instance, the length of arrays can be dynamically changed. Array elements may be of different types and they can be added with high indexes which may create undefined elements in an array. In TAJS, array elements that are assigned values separately are tracked individually. For those elements that are not in the domain of the map, presented previously, TAJS has two special properties default_index and default_other. The default_index approximates the values of all property names that match UIntString and default_other approximates the values of all property names that match NotUIntString. In TAJS$_{\mathrm{oct}}$, we use intervals to abstract the set of array indexes and the set of values they might hold. For instance, if we have the following program

```
var A;
A = populateArray(0, 5, A);
A = populateArray(10, 15, A);

function populateArray(x, y, A) {
   var i;
   for(i=x ; i<=y ; i++)
     A[i]= 2i;
   return A;
}
```

The array A will be described by the following abstract object: $\{[0,5] \to [0,10], [10,15] \to [20,30]\}$. The element $([0,5] \to [0,10])$, for example, indicates that integers from 0 to 5 are possible indexes of A and values at these array indexes are numbers between 0 and 10. These numbers are either unsigned integers or floating-point numbers as determined by the type domain.

Each property name that is assigned a numerical value individually is represented by a dimension in the octagon. In addition, each interval representing a set of array indexes is also represented by a dimension in the octagon. In the previous example, the element $A[0,5]$ will be mapped to a dimension that represents the possible values for indexes from 0 to 5. The element $A[10, 15]$ will be mapped to another dimension that represents the possible values for indexes from 10 to 15.

6.4 Objects

An object in JavaScript is a collection of properties and functionality. An object property could be of any type in a similar way to variables and may be added or removed dynamically. TAJS incorporates recency abstraction [5] which is a solution that allows an abstract object to describe an unlimited number of concrete objects and permits strong updates on objects. We took advantage of this solution to perform strong updates on objects. When an object is created, the octagon is extended with a dimension corresponding to each property. Removing any property causes the related dimension to be removed from the octagon. For instance, if we have

```
function A(a)
{ this.a = a; }
...
var A1 = new A(20);
var A2 = new A(30);
```

Then each object property corresponds to a dimension in the octagon as follows $O = \{l_1.a = 20, l_2.a = 30, l_1.a + l_2.a = 50\}$ where l_1 is the allocation site of the object $A1$ and l_2 is the allocation site of the object $A2$.

6.5 Prototypes

Prototype is a fundamental concept in JavaScript. JavaScript uses prototype objects to model inheritance. Every object has a prototype from which other objects can inherit methods and properties. The JavaScript prototype property can be used to add a new property to an existing prototype and hence all objects that inherit from that prototype will have access to the new property. Fields on the prototype will be shared between instances. When an object is created, all properties of the object including properties inherited through the prototype will be represented in the octagon and each object has its own properties. A property on the prototype that is shared between instances will be mapped to a single dimension in the octagon that can be accessed by all instances. In the following code

```
function Person(age)
{ this.age = age; }
...
var p1 = new Person(20);
var p2 = new Person(30);
Person.prototype.height = 170;
```

both $p1$ and $p2$ are instances of $Person$ and each has its own copy of age. The octagon will be extended with two dimensions to abstract $p1.age$ and $p2.age$. The property $height$ behaves like a static field and therefore will be mapped to one dimension in the octagon that can be accessed by all instances of $Person$.

7 Evaluation

We implemented the new abstract domains on top of TAJS for the full language of JavaScript. In our experiments, we evaluate precision and performance of TAJS_{oct} and we compare the

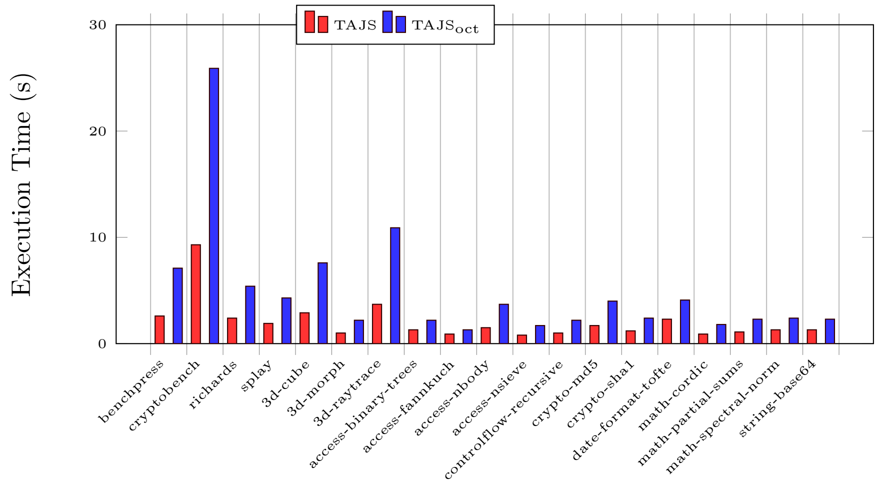

Figure 3: Comparison on analysis time (seconds) between TAJS and $TAJS_{oct}$

results with the original one. The test platform is a MacOS Sierra with 2.6 GHz Intel Core i5 processor and 8GB memory.

Benchmarks. TAJS contains a collection of benchmarks. Some of the benchmarks included are the standard SunSpider (26 tests) and V8 programs (196 tests), Google programs (5 tests), ChromeExperiements (35 tests) and JS1k programs (266 tests). We ran our analysis on these benchmarks and we have slightly modified some of them by inlining some conditions and assertions. However, we have chosen some of the most used benchmarks to present our results which are shown in Fig. 3.

Performance. The results in Fig. 3 show that there is an increase in the time needed to complete the tests in $TAJS_{oct}$. Unlike constant propagation domains which have a linear time complexity, the time complexity of abstract operators of the octagon domain in our analysis is $O(n^3)$. The octagon domain is also combined with type abstract domain in order to infer more precise information about variables than TAJS. Thus, the increase in analysis time is expected. The results also show a higher execution time in the benchmarks that do heavier numerical computation compared to those with less numerical computation. The average percentage of the increase is roughly 147%.

Precision. TAJS generates warnings of possible type errors. Some of these warnings are false due to over-approximation. The over approximation in TAJS causes some infeasible paths wrongly identified as feasible leading to further over approximation. The gain in precision and the number of infeasible paths detected in the new analysis has led to the reduction of the number of false warnings. In addition, the imprecision in TAJS causes it to report inaccurate information about variables in some cases, such as a possible NaN or undefined values. The new analysis in many cases can eliminate such false positives that may result in some unreported errors. Fig. 4 show the reduction in false positives in $TAJS_{oct}$ compared to TAJS. The average percentage of the reduction in false positives is roughly 9.5%.

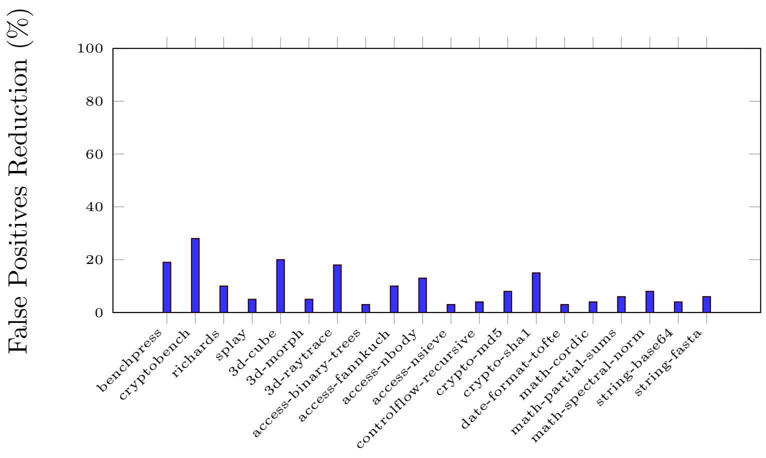

Figure 4: Decrease in false positives in TAJS_{oct}

8 Conclusion

JavaScript has dynamic features that make it hard to reason about. This makes analysis tools both necessary and hard to develop. We enhanced TAJS, concentrating on the numerical domain by incorporating the octagon and type domains as abstract domains for number values. Our analysis can detect some runtime errors such as division by zero and violation of user-defined assertions. Results show that the output from TAJS_{oct} is much more precise as it gives relational information between numerical program variables.

References

[1] Agostino Cortesi and Matteo Zanioli. Widening and narrowing operators for abstract interpretation. *Computer Languages, Systems & Structures*, 37(1):24–42, 2011.

[2] Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of programs. In *Proceedings of the second International Symposium on Programming*, pages 106–130. Paris, 1976.

[3] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

[4] Patrick Cousot and Radhia Cousot. Comparing the galois connection and widening/narrowing approaches to abstract interpretation. In *Programming Language Implementation and Logic Programming*, pages 269–295. Springer, 1992.

[5] Phillip Heidegger and Peter Thiemann. Recency types for dynamically typed object-based languages. In *International Workshops on Foundations of Object-Oriented Languages*. FOOL, 2009.

[6] Simon H. Jensen, Anders Møller, and Peter Thiemann. Type analysis for JavaScript. In *Static Analysis*, pages 238–255. Springer, 2009.

[7] Simon H. Jensen, Anders Møller, and Peter Thiemann. Interprocedural analysis with lazy propagation. In *Static Analysis*, pages 320–339. Springer, 2011.

[8] Vineeth Kashyap, Kyle Dewey, Ethan A. Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. JSAI: A static analysis platform for Javascript. In *22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 121–132. ACM, 2014.

[9] Hongki Lee, Sooncheol Won, Joonho Jin, Junhee Cho, and Sukyoung Ryu. SAFE: Formal specification and implementation of a scalable analysis framework for ECMAScript. In *International Workshop on Foundations of Object-Oriented Languages (FOOL)*, 2012.

[10] Francesco Logozzo. *Cibai: An Abstract Interpretation-Based Static Analyzer for Modular Analysis and Verification of Java Classes*, pages 283–298. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[11] Francesco Logozzo and Herman Venter. RATA: rapid atomic type analysis by abstract interpretation–application to Javascript optimization. In *Compiler Construction*, pages 66–83. Springer, 2010.

[12] Antoine Miné. Relational abstract domains for the detection of floating-point run-time errors. In David Schmidt, editor, *Programming Languages and Systems*, pages 3–17, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[13] Antoine Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, Mar 2006.

[14] Peter Thiemann. Towards a type system for analyzing javascript programs. In *Programming Languages and Systems*, pages 408–422. Springer, 2005.

[15] Astrid Younang, Lunjin Lu, and Nabil Almashfi. Static checking of array objects in javascript. In *4th Tools and Methods of Program Analysis International Conference (TMPA)*, 2017.