The Good, the Bad, and the Ugly: An Empirical Study of Implicit Type Conversions in JavaScript

Michael Pradel[1] and Koushik Sen[2]

1   **TU Darmstadt**
    Department of Computer Science
    Germany
    `michael@binaervarianz.de`
2   **University of California, Berkeley**
    EECS Department
    USA
    `ksen@berkeley.edu`

─── **Abstract** ───

Most popular programming languages support situations where a value of one type is converted into a value of another type without any explicit cast. Such implicit type conversions, or type coercions, are a highly controversial language feature. Proponents argue that type coercions enable writing concise code. Opponents argue that type coercions are error-prone and that they reduce the understandability of programs. This paper studies the use of type coercions in JavaScript, a language notorious for its widespread use of coercions. We dynamically analyze hundreds of programs, including real-world web applications and popular benchmark programs. We find that coercions are widely used (in 80.42% of all function executions) and that most coercions are likely to be harmless (98.85%). Furthermore, we identify a set of rarely occurring and potentially harmful coercions that safer subsets of JavaScript or future language designs may want to disallow. Our results suggest that type coercions are significantly less evil than commonly assumed and that analyses targeted at real-world JavaScript programs must consider coercions.

1   Introduction

In most popular programming languages, a value has a type that determines how to interpret the data represented by the value. To change the way a value is interpreted, programmers can convert a value of one type into a value of another type, called *type conversion*. In addition to explicit type conversions, or casts, many languages perform implicit type conversions, called *type coercions*. Type coercions are applied by the compiler, e.g., to transform an otherwise type-incorrect program into a type-correct program, or by the runtime system, e.g., to evaluate expressions that involve operands of multiple types. Both statically typed and dynamically typed languages use type coercions. For example, C and Python coerce numeric values from integer types to floating point types, and vice versa, and Java coerces instances of a subtype into an instance of a supertype. Some dynamically typed languages, such as JavaScript, make even wider use of type coercions.

Type coercions are a highly controversial language feature. On the one hand, they enable programmers to write concise code. On the other hand, type coercions can obfuscate a program, because the source code does not clarify which types of values a piece of code operates on, and even hide errors, because an unintended operation or an unintended loss of information may remain unnoticed.

This paper focuses on JavaScript, a language notorious for its heavy use of type coercions. Many operations that are considered illegal in languages with a stricter type system are legal in JavaScript. The reason is that JavaScript avoids throwing runtime type mismatch exceptions as much as possible so that programs can tolerate errors as much as possible. Although the rules for coercing types are well defined [10], even expert JavaScript developers struggle to fully comprehend the behavior of some code. For illustration, consider the following rather surprising examples:

- `"false" == false` evaluates to `false`, but `"0" == "false"` evaluates to `true`.
- `new String("a") == "a"` and `"a" == new String("a")` evaluate to `true`, but
  `new String("a") == new String("a")` evaluates to `false`.
- `[] << "2"` evaluates to `0`, `[1] << "2"` evaluates to `4`, and `[1,2] << "2"` evaluates to `0`.

Because type coercions can lead to such surprising behavior in JavaScript, it is common to assume that coercions are error-prone and therefore rarely used in practice. Based on these assumptions, existing static type inference and checking approaches for dynamically typed languages do not support type coercions at all [25, 2] or allow only a limited set of coercions [11, 4]. For example, Chugh et al. prohibit implicit coercions since they "often lead to subtle programming errors" [2].

While such assumptions about coercions are common, little is currently known about how type coercions are used in practice. This paper addresses this problem and asks the following research questions:

- RQ 1: How prevalent are type coercions in JavaScript, and what are they used for?
- RQ 2: Are type coercions in JavaScript error-prone?
- RQ 3: Do type coercions in JavaScript harm code understandability?

Answering these questions provides insights that enable informed decision making in at least three situations. First, developers of static and dynamic analyses benefit from the answers when deciding about how to handle coercions. Second, developers of safer subsets of JavaScript, such as *strict mode* [10], can use the answers to estimate the number of changes that a particular subset of JavaScript imposes on existing programs. Third, designers of future languages can benefit from the answers to decide whether and how to include type coercions.

To address these questions, this paper presents an empirical study of type coercions in real-world JavaScript web applications and in popular benchmark suites. As a result of the study, we can substantiate and refute several pieces of conventional wisdom, respectively. Our results include the following findings:

- Type coercions are widely used: 80.42% of all function executions perform at least one coercion and 17.74% of all operations that may apply a coercion do apply a coercion, on average over all programs. (RQ 1)
- In contrast to coercions, explicit type conversions are significantly less prevalent. For each explicit type conversion that occurs at runtime, there are 269 coercions. (RQ 1)
- We classify coercions into harmless and potentially harmful coercions, and we find that 98.85% of all coercions are harmless and likely to not introduce any misbehavior. (RQ 1)
- A small but non-negligible percentage (1.15%) of all type coercions are potentially harmful. Future language designs or restricted versions of JavaScript may want to forbid them.

For today's JavaScript programs, a runtime analysis can report these potentially harmful coercions. (RQ 2)

- Out of 30 manually inspected potentially harmful code locations, 22 are, to the best of our knowledge, correct, and only one is a clear bug. These results suggest that the overall percentage of erroneous coercions is very small. (RQ 2)
- Most code locations with coercions are monomorphic (86.13%), i.e., they always convert the same type into the same other type, suggesting that these locations could be refactored into explicit type conversions for improved code understandability. (RQ 3)
- Most polymorphic code locations (93.79%), i.e., locations that convert multiple different types, are conditionals where either some defined value or the `undefined` value is coerced to a boolean. (RQ 3)
- JavaScript's strict and non-strict equality checks are mostly used interchangeably, suggesting that refactoring non-strict equality checks into strict equality checks can significantly improve code understandability. (RQ 3)

All data gathered for this study and the full details of our results are available for download:

<div align="center">

`http://mp.binaervarianz.de/ecoop2015`

</div>

In summary, this paper contributes the following:

- The first in-depth study of type coercions in real-world JavaScript programs.
- Empirical evidence that contradicts the common assumption that type coercions are rarely used and error-prone. Instead, we find that coercions are highly prevalent and mostly harmless.
- A classification of coercions into harmless and potentially harmful that may guide the development of safer subsets of JavaScript and future language designs.

2 Methodology and Subject Programs

To answer the research questions from the introduction, we perform a study of type coercions in real-world JavaScript programs. We dynamically analyze an execution of each program to record all runtime events that may cause a coercion (Section 2.1). Based on these data, we perform a set of offline analyses that summarize the runtime data into representations that allow for answering the research questions.

2.1 Dynamic Analysis

To gather runtime information about coercions, we instrument the program to intercept all runtime events that may cause a coercion or an explicit type conversion:

- *Unary and binary operations.* For unary and binary operations, the analysis records the source code location, the operator, the types of the input(s) and the output of the operation, and an abstraction (defined below) of the runtime values of the input(s) and the output of the operation.
- *Conditionals.* For each conditional evaluated during the execution, the analysis records the code location, the type and the abstracted runtime value of the evaluated expression, and the boolean value into which the expression gets coerced.
- *Function calls.* For each function call, the analysis checks whether the called function is any of the built-in functions that explicitly convert values from one type to another, i.e., `Boolean`, `Number`, and `String`. In this case, the analysis records the code location, as well as the types and the abstracted values of the function argument and the return value.

We abstract runtime values as follows: For booleans, `undefined`, `null`, and `NaN` (not a number), the analysis stores the value. For other numbers, it stores whether the value is zero or non-zero. For strings, it stores whether the string is empty or non-empty. For objects, including arrays, it stores whether the object has any own properties, i.e., whether it is an empty object or array. Furthermore, the analysis records whether an object has `valueOf` and `toString` methods that differ from the default implementation inherited from `Object` and, if an object has such a `valueOf` method, the type of `valueOf`'s return value. This extra type information allows us to identify values that explicitly define how to coerce them into another type.

Our implementation builds upon Jalangi [22], which instruments the JavaScript source code so that the analysis can intercept all necessary runtime events. For programs that execute on Node.js, we instrument the program on the file system. For web applications, we modify the Firefox browser so that it intercepts and instruments all JavaScript code before executing the code. The dynamic analysis creates a file that summarizes all recorded information for a program, and we analyze this file offline, i.e., after the execution.

For the purpose of this study, dynamic analysis has several benefits over a comparable static analysis. First, statically determining whether a code location coerces a type is impossible in general due to the highly dynamic nature of JavaScript. We believe that statically overapproximating potential coercions may skew the study results. Second, dynamic analysis enables us to reason about concrete runtime values, which is important for the qualitative part of our study, where we manually inspect coercions to determine whether coercions are harmful. Finally, dynamic analysis enables us to quantify how often coercions occur at runtime. On the downside, dynamic analysis misses coercions on paths that are not executed. We discuss implications of this limitation in Section 5.

2.2   Subject Programs

Our study considers three kinds of programs: the SunSpider benchmarks, the Octane benchmarks, and the top 100 most popular web sites according to Alexa[1]. For each benchmark, we analyze an execution of the benchmark in its default setup. For each web site, we load the start page and analyze all JavaScript code that gets executed, including code loaded from third-party libraries. In total, the study considers 138,979,028 runtime events from 132 programs. These events are generated by 321,711 unique source code locations.

In addition to the main research questions of this paper, this setup also allows us to address the question how accurately the benchmark suites, which are widely used to evaluate JavaScript engines, represent the type coercion-related behavior of real-world web applications.

3   Classification of Type Coercions

In this section, we propose a classification of all type coercions that may occur in JavaScript into likely harmless and potentially harmful coercions. This classification may serve three purposes. First, we use it to approximate the harmfulness of coercions in practice by classifying the coercions we observe in real-world programs. Second, the classification may provide a basis for defining a safer subset of JavaScript that forbids or warns about potentially harmful coercions. Such a subset may be enforced through runtime checks, similar to JavaScript's strict mode [10]. Third, the classification may guide future language designs that want to

---

[1] `http://www.alexa.com/topsites`, accessed on July 16, 2014

allow harmless coercions for conciseness but disallow potentially harmful coercions. The results of our study will help approaches of the second and third direction by providing empirical data about how often different kinds of coercions occur, i.e., how much code one would break by disallowing them.

Since developers may purposefully exploit the behavior of any type coercion, there is no clear-cut definition of when a coercion constitutes an error. The proposed classification is based on our own experience with JavaScript, on reports of the experience of others, e.g., in web forums, and on a comparison with other programming languages. We classify a coercion as *potentially harmful* if its semantics deviates from what is common in other, more strongly typed languages, such as C, Java, Python, or Ruby, if the operation that triggers the coercion has no intuitive meaning, or if the rules that determine which coercion to apply are very complex. We classify all other type coercions as harmless. The remainder of this section presents and illustrates our classification, which is summarized in Table 1.

3.1 Terminology

JavaScript has six basic types: The three primitive types boolean, number, and string, the special, single-value types `undefined` and `null`, and the object type, which includes arrays and functions. To simplify our classification, we use the following additional terms.

- A *quasi-number* is a primitive number or an object that defines a `valueOf` method that returns a primitive number. Developers can use `valueOf` to specify how to coerce an object, and the execution environment calls the method whenever the language requires a coercion.
- A *quasi-string* is a primitive string or an object that defines a `valueOf` method that returns a primitive string.
- A *wrapped primitive* is an object created with one of the built-in wrappers, `new Boolean()`, `new Number()`, and `new String()`.
- An *empty object* is an object without any own properties, e.g., the result of evaluating the object literal expression `{}`.
- An *empty array* is an array with length zero, e.g., the result of evaluating the array literal expression `[]`.
- A *defined value* is every value except for `undefined` and `null`.

3.2 Conditional-related Coercions

In JavaScript, values of all types may be used as conditions. Furthermore, all types may occur as the operand of the logical negation operator `!` and as the operands of the binary logical operators `&&` and `||`. The semantics are straightforward: All objects, all strings except the empty string, and all numbers except zero and `NaN` coerce to `true`. Note that all objects include objects whose `valueOf` returns `false`, empty arrays, and empty objects. All other values, including `undefined` and `null`, coerce to `false`. Because of these rather simple semantics and because using arbitrary types in conditionals is very common in JavaScript, we consider these coercions as harmless.

As an exception to the above classification, we classify wrapped primitives used as conditions or as operands of `!`, `&&`, and `||` as potentially harmful. Because all objects, including wrapped primitives, coerce to `true`, the semantics of wrapped primitives in conditionals differs from their primitive counterparts, which may surprise developers. For example, the wrapped primitive `new Boolean(false)` coerces to `true`. Popular guidelines [3] suggest to avoid wrapped primitives altogether.

Operation	Type of operands	Example	Comment	Class.
Conditional:				
-	Wrapped primitives	`if (new Boolean(false))`	Behavior differs from primitives	✘
-	All other types	`if (someNumber)`	Coerced to boolean	✔
Unary operations:				
`+, -, ~`	All except quasi-numbers	`-"abc"`	Coerced to number	✘
`!`	Wrapped primitives	`!(new Boolean(false))`	Behavior differs from primitives	✘
`!`	All except wrapped primitives	`!someNumber`	Coerced to boolean	✔
Binary operations:				
`-, *, /, %`	All except quasi-numbers	`"abc" * false`	Meaningful only for numbers	✘
`<<, >>, >>>`	All except quasi-numbers	`{} << 23`	Meaningful only for numbers	✘
`+`	Quasi-string and `undefined`, or quasi-string and `null`	`var x; x+="abc"`	Result contains `"undefined"` or `"null"`	✘
`+`	Quasi-string and a defined value	`"Names: " + arrayOfNames`	Common to construct strings	✔
`+`	Two non-quasi-strings	`false + [2,3]`	Confusing semantics	✘
`<, >, <=, >=`	All except two quasi-numbers and two quasi-strings	`[1,2] < function f() {}`	Meaningful only for numbers and strings	✘
`==, !=`	All types, unless both types are the same, or one is `undefined` or `null`	`0 == "false"`	Confusing semantics	✘
`==, !=`	One value is `undefined` or `null`	`someObject != null`	Common in conditionals	✔
`|, ||`	Left type is `undefined`, right value is `0`	`x = (x | 0) + 1`	Common pattern to initialize counters	✔
`&, |, ^`	All types, unless both are quasi-numbers, or counter initialization pattern from above	`[1,2] & "abc"`	Meaningful only for numbers	✘
`&&, ||`	At least one wrapped primitive	`new Number(0) || new Boolean(false)`	Behavior differs from primitives	✘
`&&, ||`	All except wrapped primitives	`someNumber && someBoolean`	Common in conditionals	✔

■ **Table 1** Classification of type coercions into likely harmless (✔) and potentially harmful (✘).

3.3   The Plus Operator

The semantics of the `+` operator is defined for numbers, where it means addition, and for strings, where it means concatenation. If any value that is neither a number nor a string is given to `+`, JavaScript coerces the value either into a primitive number or into a primitive string. Then, it applies string concatenation if at least one operand is a string and addition otherwise. The rules for deciding whether a value gets coerced into a number or a string are somewhat intricate. For most but not all values, JavaScript attempts to coerce the value into a number by calling `valueOf` and falls back on coercing into a string by calling `toString`. We refer to [10] for a full description of the rules.

Given that `+` has easy to understand semantics when being applied to two quasi-numbers or to two quasi-strings, we classify coercions that happen in these cases as harmless. We also classify coercions as harmless if they result from combining a quasi-string with any defined value because developers commonly use this pattern to concatenate strings. For example, `"Names:  " + arrayOfNames` causes a harmless coercion of an array into a string representation of the array. In contrast to these harmless coercions, we classify all other coercions triggered by `+` as potentially harmful because their semantics differ from more strongly typed languages. For example, the statements `var x; x+="abc";`, which yield the string `"undefinedabc"` instead of the probably expected `"abc"`, and the expression `false+{}`, which yields "false[object Object]", are classified as potentially harmful.

3.4   Arithmetic and Bitwise Operators

The unary `+` and `–` operators coerce their operand to a number. The arithmetic operators `–`, `*`, `/`, `%`, and the bitwise shift operators `<<`, `>>`, and `>>>` are defined for numbers, and applying them to any types except for two numbers triggers a coercion to number. Similar to the arithmetic operators, the unary bitwise `~` operator and the binary bitwise operators `&`, `|`, and `^` are defined for 32-bit integers. Applying these operators to any other values leads to a coercion of the operands into 32-bit integers.

Because all these operations are meaningful only for numbers and values that coerce into a number, we consider them as harmless when being applied to quasi-numbers, and as potentially harmful otherwise. For example, we consider the following operations as potentially harmful:

- `-"abc"`, which results in `NaN`
- `[1,2] & "abc"`, which yields `0`
- `{} << 23`, which yields `0`

The above classification considers coercing a string to a number in an arithmetic operation as potentially harmful. The reason is that one arithmetic operator, `+`, has a different semantics than the other arithmetic operators, which may easily confuse developers: `23 - "5"` is interpreted arithmetically and yields `18`, but `23 + "5"` is interpreted as string concatenation and yields `"235"`.

As an exception to the above classification, we consider a common code idiom for initializing counter variables, e.g., `x = (x | 0) + 1`. This idiom initializes `x` to zero when the code is executed for the first time, and it increments `x` otherwise. That is, at the first execution, the idiom coerces `undefined` to `0`. Because this idiom has clear semantics and is commonly used, we classify coercions caused by this pattern as harmless.

3.5   Relational Operators

The semantics of the relational operators `<`, `>`, `<=`, and `>=` is defined for numbers (numeric comparison) and for strings (lexicographic order). JavaScript coerces any pairs of values that contain a non-number or a non-string to either a pair of numbers or a pair of strings, and then applies the respective operation. The rules for such coercions are the following: At first, JavaScript coerces any non-primitive into a primitive, where a `valueOf` method that returns a number is preferred over a `toString` that returns a string. Then, JavaScript coerces any remaining non-numbers into numbers, unless both primitives are strings, in which case the lexicographic order is computed.

Because relational operators have an intuitive semantics for pairs of numbers and for pairs of strings, we classify coercions that occur when combining two quasi-numbers or two quasi-strings as harmless. In contrast, we classify all other coercions as potentially harmful, because relational operations have no meaningful semantics for other types. For example, we consider `[1,2] < function foo() {}`, which yields `true`, as potentially harmful.

3.6   Equality Operators

JavaScript provides two kinds of (in)equality operators: the strict operators `===` and `!==`, and the non-strict operators `==` and `!=`. The strict operators never apply any type coercions; instead, they consider any two values of different types as unequal. In contrast, their non-strict counterparts coerce operands to evaluate whether they may be considered equal despite having different types. Because the coercion rules for non-strict equality operations are rather complex and sometimes unintuitive (see [10] for full details), guidelines [3] recommend to avoid non-strict equality operations. Examples for the somewhat surprising behavior of non-strict equality operations include:

- `"" == 0` yields `true` but `"false" == 0` yields `false` because comparing a string and a number leads to coercing the string into a number, where `""` is coerced into `0` and `"false"` is coerced into `NaN`.
- `true == "true"` yields `false` but `true == "1"` yields `true` because comparing a boolean and any non-boolean leads to coercing the boolean into a number, which coerces `true` to `1`.
- Both `new Number(5) == 5` and `5 == new Number(5)` yield `true` but `new Number(5) == new Number(5)` yields `false` because non-strict equality is non-transitive.

Because of their rather confusing semantics, we consider type coercions caused by non-strict equality operations as potentially harmful, including all the examples given above. The only exception to this classification are comparisons of arbitrary values with `undefined` or `null`. Because `undefined` and `null` essentially mean the same in a non-strict comparison, such comparisons are commonly used to check for defined values. For example, we classify the following check as harmless: `someObject != null`.

4   Type Coercions in the Wild

4.1   RQ 1: Prevalence of Type Coercions

In the following, we address the question how prevalent type coercions are in real-world JavaScript programs.
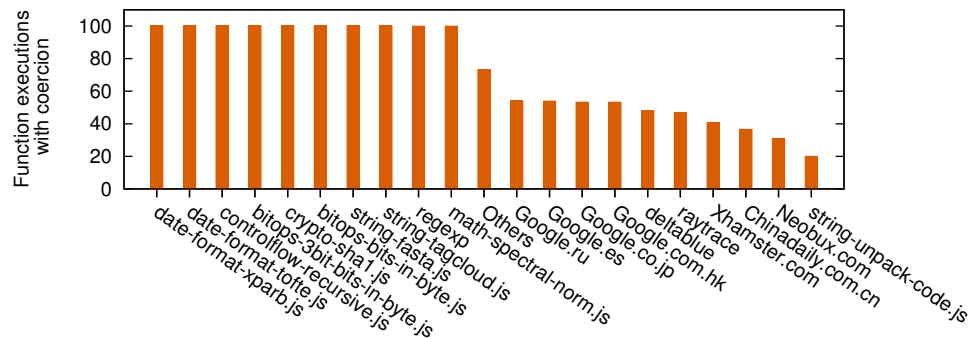
■ **Figure 1** Percentage of function executions with at least one type coercion (top 10 and bottom 10 programs only).

4.1.1 Function Executions With At Least One Coercion

As a measure of the prevalence of type coercions, we assess during how many function executions at least one coercion occurs:

▶ **Definition 1** (Function executions with coercion, FEC)**.** The percentage FEC of function executions with a coercion is the number of function executions where at least one type coercion occurs between entering the function and exiting the function, excluding the execution of the function's callees, divided by the total number of function executions.

For example, executing the following program yields a FEC of 50% because `f()` does not perform any coercion, but calling `g()` triggers the coercion in line 6.

```
1  function f() {
2    g();
3    return 5 + 1;
4  }
5  function g() {
6    return 5 + true;
7  }
8  f();
```

Over all programs we analyze, the FEC is 80.42%. This number indicates that coercions are a highly prevalent phenomenon that cannot be ignored when analyzing real-world JavaScript programs.

Figure 1 shows the FEC for a subset of all programs. The figure includes the ten programs with the highest percentage, the ten programs with the lowest percentage, and the average percentage of all other programs ("Others"). These results show that even programs with relatively few coercions still perform a non-negligible number of coercions. The figure excludes programs with less than 100 function executions because the FEC is not representative for these programs. In particular, the figure excludes several SunSpider benchmarks that perform less than 10 function executions and for which the FEC is 100%.

4.1.2 Coercions versus Non-coercions

As another measure of how prevalent coercions are, we compute how often an operation that could lead to a coercion does coerce one or more values:

(a) Dynamic occurrences.

(b) Static occurrences.
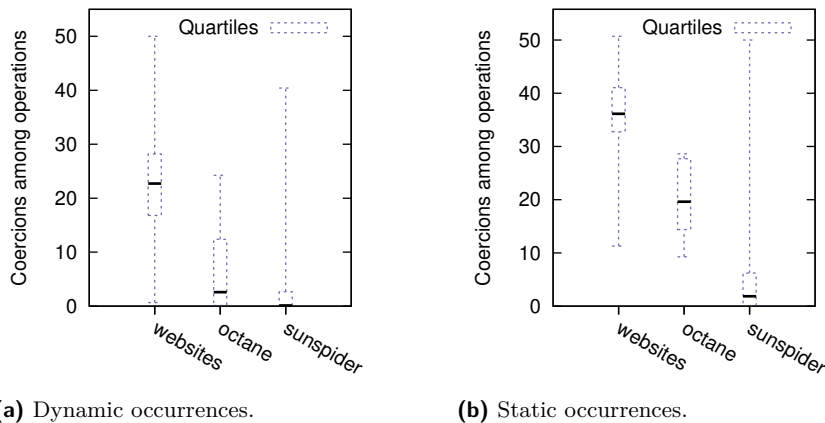
Figure 2 Prevalence of type coercions as percentage over all operations where type coercions may occur.

▶ **Definition 2** (Coercions among operations, CAO)**.** The percentage CAO of coercions among operations is the number of operations that perform a coercion divided by the total number of executed unary and binary operations and evaluated conditionals that could perform a coercion.

Figure 2 shows the CAO for the three groups of programs that we analyze. For each group, the figure shows the mean CAO over all programs from the group, the upper and lower quartiles, and the minimal and maximal CAO. We show two variants of the CAO measure. Figure 2a is based on the dynamic frequency of operations, i.e., each dynamic occurrence of an operation counts. In contrast, Figure 2b is based on static code locations, i.e., each static code location counts at most once.

The results reveal two interesting properties. First, type coercions occur in a non-negligible fraction of all operations that may cause coercions. For web sites, 36.25% of all code locations that may coerce values indeed do it in the analyzed executions. Second, type coercions are significantly more prevalent in web sites than in the SunSpider and Octane benchmarks. These benchmark suites have been criticized to be unrepresentative for real-world JavaScript programs [21, 19], and our study confirms this observation for type coercions.

Figure 3 shows the CAO for the top 10 and bottom 10 programs. On many popular web sites, such as Yahoo.com and Apple.com, more than a third of all operations that could perform a coercion do perform a coercion. The 10 programs with the smallest CAO are all from the SunSpider or Octane benchmarks, which again shows that these benchmarks fail to accurately represent real-world web applications. The figure excludes programs with less than 100 observations because measuring their CAO does not provide representative results.

4.1.3   Kinds of Type Coercions

To better understand why coercions are so prevalent in real-world JavaScript programs, we analyze what kinds of coercions occur. Figure 4 shows the most and the least prevalent kinds of type coercions. The horizontal axis clusters similar kinds of coercions. For example, "number in conditional" means that a value of type `number` is coerced into a `boolean` because it occurs in a conditional, and "+-˜ null" means that one of the arithmetic operators `+`, `-`, or `˜` is applied to `null`, which leads to a coercion into the number zero.
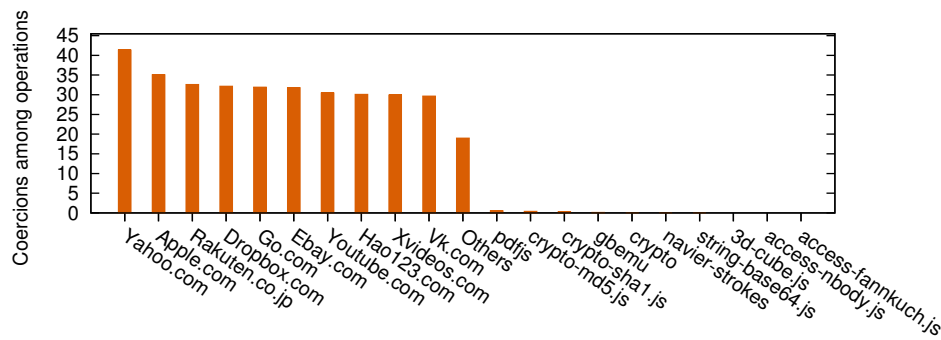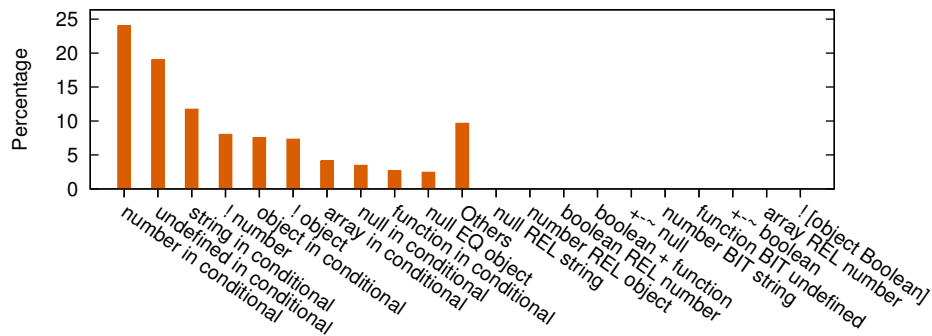
(a) Dynamic occurrences.
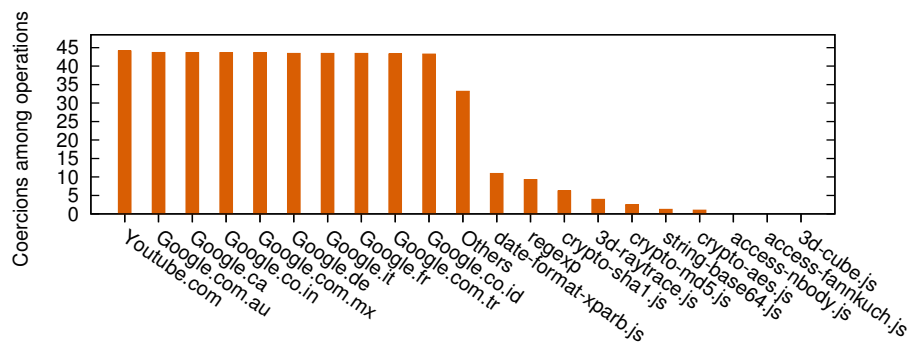
(b) Static occurrences.

Figure 3 Programs with the highest and lowest prevalence of type coercions (measured as percentage over all operations where type coercions may occur).

The figure shows that conditionals and logical negations, which are typically used in conditionals, are the most prevalent kinds of coercion. Overall, coercions that result from conditionals or from operations that are typically used in conditionals (`!`, `&&`, and `||`) account for 93.01% of all coercions. This result suggests that analyses of JavaScript, such as type inference and checking approaches, should at least consider these kinds of coercions because they occur frequently in practice.

4.1.4 Implicit versus Explicit Type Conversions

As an alternative to coercions, JavaScript developers can explicitly convert values from arbitrary types into booleans, numbers, and strings using the built-in functions `Boolean`, `Number`, and `String`. We measure how prevalent such explicit type conversions are and compare their prevalence to coercions. In total, we observe 20,407 explicit type conversions during the execution of all programs, and 5,497,545 coercions. That is, for every explicit type conversion that occurs, there are 269 implicit type conversions. We conclude that explicit conversions are used significantly less frequently than coercions. A possible explanation is that developers prefer the conciseness of coercions over the potentially increased code understandability through explicit conversions.

(a) Dynamic occurrences.
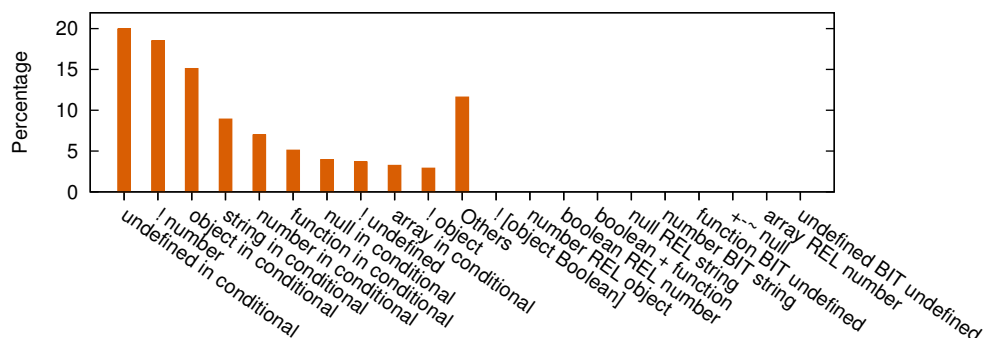
(b) Static occurrences.

Figure 4 Prevalence of different kinds of type coercions as percentage over all type coercions (top 10 and bottom 10 only). Horizontal axes: BIT, EQ, and REL mean bitwise, equality, and relational operations, respectively.

4.2 RQ 2: Harmfulness of Type Coercions

We address the question whether type coercions are error-prone in two ways. First, we measure how many type coercions are harmless and potentially harmful according to the classification from Section 3. Second, we manually inspect a sample of the potentially harmful type coercions to assess whether their behavior is intended or erroneous.

4.2.1 Harmless versus Potentially Harmful Coercions

Given the classification from Section 3, Figure 5 shows how many of all observed type coercions are potentially harmful. For most programs, a relatively small percentage of coercions is potentially harmful. For example, on average over all websites, only 1.25% of all dynamic occurrences of coercions are potentially harmful. The average over the coercions of all programs is 1.15%. Some of the SunSpider benchmarks are outliers in Figure 5, with up to 99.98% potentially harmful coercions. By manually inspecting these benchmarks we find that they indeed perform various potentially harmful coercions, which lead to a high percentage because these benchmark programs are relatively small.

Figure 6 shows the ten applications that have the highest and lowest percentage of poten-

(a) Dynamic occurrences.   **(b)** Static occurrences.
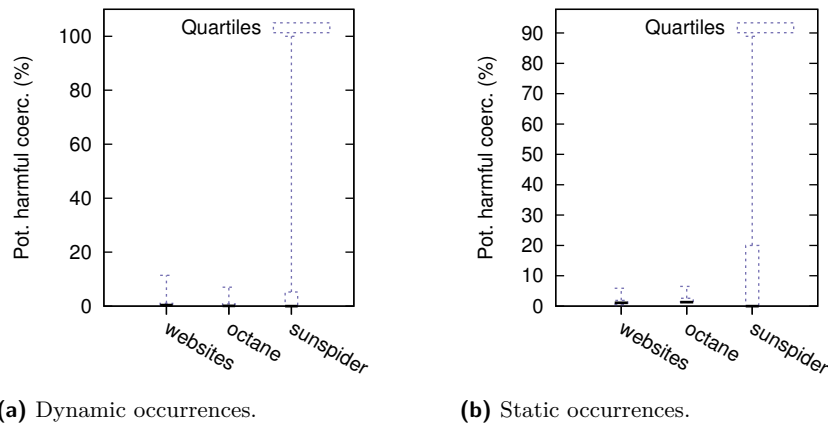
Figure 5 Percentage of potentially harmful coercions over all coercions.

tially harmful coercions, respectively. Apart from the outliers in the SunSpider benchmarks discussed above, all programs have less than 12% potentially harmful coercions (for both dynamic and static occurrences).
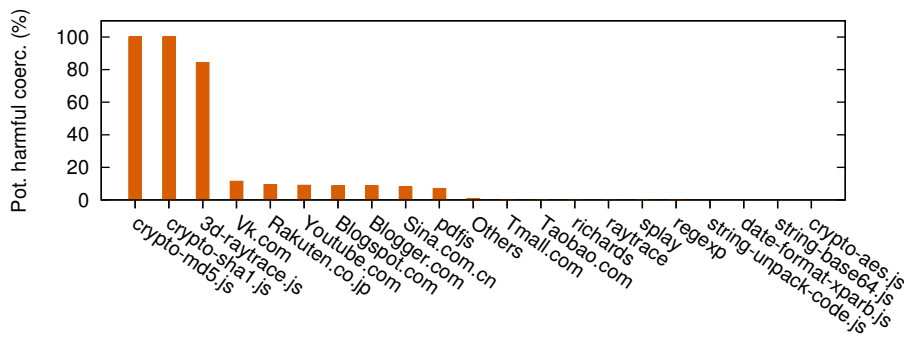
4.2.2 Kinds of Potentially Harmful Coercions

Since the number of potentially harmful coercions is non-negligible, we further analyze what kinds of potentially harmful coercions occur. Figure 7 shows the 10 most prevalent and the 10 least prevalent kinds of potentially harmful coercions. The most prevalent potentially harmful coercion (by number of static occurrences) are non-strict (in)equality checks that compare two objects of different types, which is a common source of confusion. Several prevalent kinds of potentially harmful coercions involve `undefined`, such as concatenating `undefined` with a string, which yields a string that contains `"undefined"`, and relative operators applied to `undefined` and a number, which always yields `false`. We speculate that most of these coercions are caused by an `undefined` value that accidentally propagates through the program.

Our results suggest that developing analyses that warn programmers about potentially harmful coercions is a promising line of future work, e.g., along the lines of [11]. Our study provides empirical data on how many warnings such analyses may yield, so that developers of such analyses can focus on rarely occurring, potentially harmful coercions. Another direction for future work are techniques that prevent the `undefined` value from propagating in undesired ways.

4.2.3 Binary Plus Operations

One of the most debated and potentially error-prone operators in JavaScript is the binary + operator. For example, a popular guideline [3] lists the operator as "problematic" and mentions that it is "a common source of bugs". To better understand how dangerous + is in practice, we analyze all occurrences of this operator.

Figure 8 shows what kinds of types + is applied on, including types that do not lead to any coercion. The labels on the horizontal axis also indicate whether a coercion occurs and if yes, whether we classify the coercion as potentially harmful. Most + operations apply to either two strings or to two numbers, i.e., they do not coerce any types. The most prevalent

(a) Dynamic occurrences.
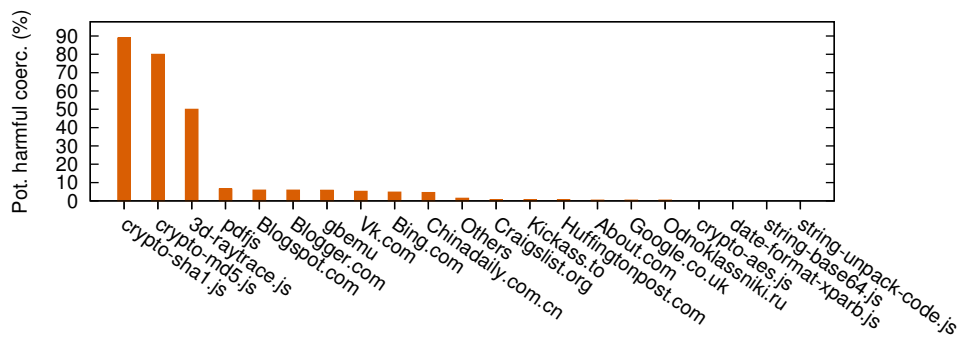
(b) Static occurrences.
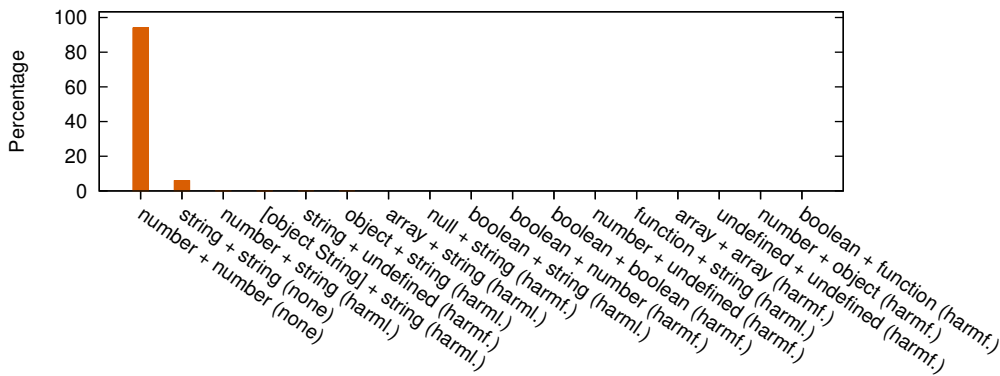
Figure 6 Programs with the highest and lowest percentage of potentially harmful coercions.

occurrence of + that coerces operands, number + string, is harmless. In total, only a very small percentage of all dynamic occurrences of + lead to a potentially harmful coercion.
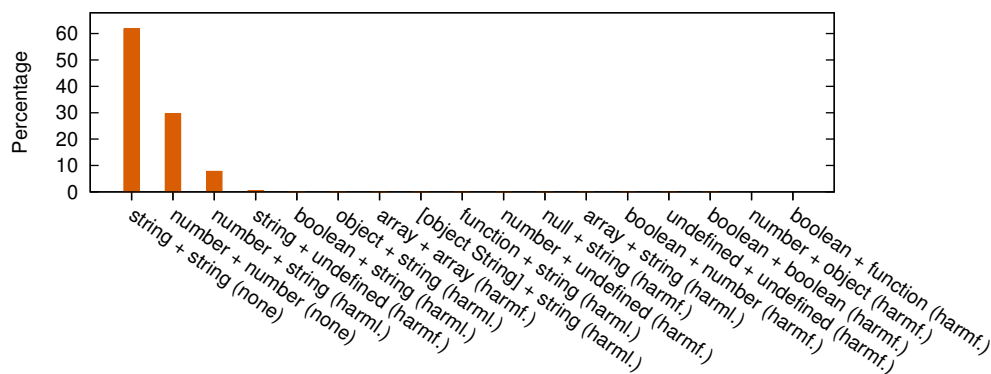
We conclude from these results that the + operator is less dangerous than commonly expected. Programmers are disciplined enough to apply + (mostly) in situations where the operation does not cause any type coercion or where it applies a harmless coercion that has obvious semantics. That said, reconsidering the semantics of + in future language designs to reduce its complexity seems to be a good idea. To deal with today's JavaScript, checking for the rarely occurring potentially harmful usages of + is a promising endeavor for static or dynamic analyses.

4.2.4   Manual Inspection of Potentially Harmful Coercions

To gain further insights into the harmfulness of coercions, we manually inspect a random sample of all potentially harmful coercions. Out of the total of 1,329 unique code locations that perform at least one potentially harmful coercion, we inspect a random sample of size 30. Inspecting these coercions is non-trivial because most web sites obfuscate and minify their JavaScript code, and because we are not familiar with the implementation of the studied programs. Therefore, we cannot provide a clear-cut classification of all inspected locations. To the best of our knowledge, out of the 30 inspected locations:

(a) Dynamic occurrences.

(b) Static occurrences.

Figure 7 Kinds of potentially harmful type coercions (top 10 and bottom 10 kinds only). Horizontal axes: ARITHM, BIT, EQ, and REL mean arithmetic, bitwise, equality, and relational operations, respectively.

- 22 are probably correct,
- 1 is a clear bug,
- 3 may be buggy, but we cannot confirm any visible misbehavior, and
- 4 are unclear.

The clear bug was part of Sina.com.cn, a popular Chinese web portal. The page contained code that encodes various environment settings, such as the current timezone or the version number of the installed Flash player, into a single number. To access the environment settings, the page provides helper functions that are supposed to compare the number, `m`, against particular bit patterns:

```
timezone: function() {
  return (m & 16384 == 16384) ? (new Date().getTimezoneOffset() / 60) : ""
},
flashVer: function() {
  if (m & 8192 != 8192) {
    return ""
  }
  ..
}
```

(a) Dynamic occurrences.

(b) Static occurrences.

Figure 8 Kinds of plus operations.

Unfortunately, these helper functions always return the same value because equality operators have precedence over the bitwise & operator. For example, in function `flashVer`, `8192 != 8192` always yields `false` and therefore `m & 8192 != 8192` always yields zero, which is coerced to `false`. This operation is classified as potentially harmful because it combines a number and `undefined` with the & operator. When we tried to reproduce the problem three days after gathering the data for this study, the corresponding code had been removed from the site.

The three locations that we classify as maybe buggy include two locations that produce `NaN` because arithmetic operations are applied to `undefined`. The other location produces a string for debugging purposes but concatenates a string with `null` into `"Source-type:  null"`, which may or may not help with debugging.

The 22 probably correct locations include several recurring patterns:

- Ten locations apply a relational or arithmetic operator to a number and a string, where the string coerces into a number. Representing numbers as strings is not recommended in general because + does not mean addition. However, the inspected code is correct because it does not use the + operator.
- Three locations concatenate a string to `undefined`, and then check whether the result matches a regular expression. This check could be implemented more efficiently, but it is correct.
- Two locations use non-strict equality checks even though they should only match if the

operands have equal types. These locations are correct but would be easier to understand if strict equality was used instead.

Our results suggest that even among the potentially harmful coercions, most coercions do not cause incorrect behavior. Instead, some developers use JavaScript's coercion semantics in unusual yet correct ways.

4.2.5 Manual Inspection of Harmless Coercions

To validate the classification from Section 3, we also inspect a random sample of size 30 of all harmless coercions. We find that all inspected coercions are indeed harmless. As expected from Figure 4, most of the coercions (26 of 30) are related to conditionals. Moreover, we identify the following recurring patterns:

- Ten coercions are from conditionals that check if a value is defined before using it. Seven of them check if an object exists before accessing its properties, three check if a function exists before calling it.
- Three coercions check if an optional function argument is defined. JavaScript supports variadic functions and optional arguments are commonly used.
- Three coercions are instances of the initialization pattern discussed in Section 3.4.
- Four coercions are due to minified code that uses `!0` and `!1` as a concise way to express `true` and `false`, respectively.

Overall, we conclude from these results that most coercions that occur in practice are harmless, which contradicts the common assumption that coercions are error-prone. We draw this conclusion for two reasons. First, even though our classification cannot rule out that some coercions classified as harmless cause errors, we believe that most JavaScript developers are aware of the semantics of the coercions that we classify as harmless. The results of manually inspecting coercions supports this assumption. Second, under the assumption that most of the analyzed programs perform the expected behavior, most of the coercions are likely to be correct, simply because coercions are very prevalent.

4.3 RQ 3: Influence on Understandability

To address the question whether and how type coercions influence the understandability of code, we analyze two particularly confusing coercion-related properties of code. First, we analyze the degree of polymorphism of code locations that apply coercions. Second, we present a detailed analysis of strict and non-strict equality checks, whose semantics often confuse developers and therefore may harm code understandability. Both analyses are proxy metrics that estimate to what degree coercions influence understandability. We leave a more detailed analysis of this question, e.g., through controlled experiments with developers [7], for future work.

4.3.1 Degree of Polymorphism

Do code locations with coercions always apply the same kind of coercion or do the types coerced at a particular location differ over time? Figure 9a shows the number of different types that are coerced at locations where we observed at least one type coercion. Most locations (86.13%) that apply a coercion always coerce values of the same types into each other, and very few locations (2.67%) apply three or more different kinds of coercions.
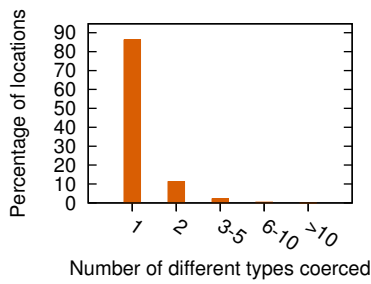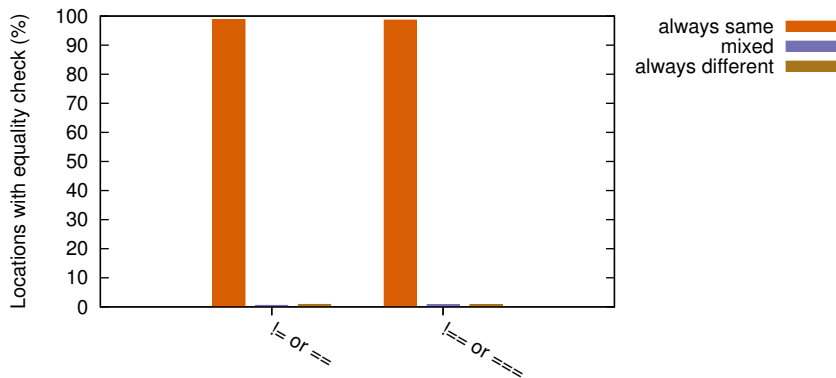
(a) Percentage of code locations that coerce a particular number of different types at different times the location is reached (only locations with coercions).

Operation	Coerced types	Locs.
conditional	[object Object], undefined	2011
conditional	string, undefined	1726
conditional	number, undefined	1034
conditional	function, undefined	746
conditional	array, undefined	685
conditional	array, null	410
!	[object Object], undefined	399
conditional	null, string	301
!	number, undefined	277
conditional	number, string	240

(b) Kinds of locations with polymorphic coercions (top 10).

Figure 9 Polymorphism of code locations that perform coercions.

Figure 10 Percentage of code locations with (in)equality checks where all observed values have always the same type, always a different type, or both, respectively.

To understand why polymorphic code locations apply multiple kinds of coercions, Table 9b list the most prevalent kinds of polymorphic coercion locations and the types that they coerce. The table shows that the main reason for polymorphic coercions are conditionals that check whether some non-boolean value is defined. This pattern is common in JavaScript and a programmer that checks whether some value is defined will expect that the check may be applied to both a defined or an undefined value.

These results suggest that most coercions do not significantly harm code understandability, at least not because of polymorphic code locations. A more detailed study on how coercions influence human understanding of code, e.g., through a human study in the style of [7] remains for future work.

4.3.2 (In)equality Checks

A particularly intricate situation where JavaScript applies type coercions are (in)equality checks with == and !=. Guidelines [3] suggest to avoid == and != altogether and to instead use their "non-evil" twins === and !==. Yet, we find that both kinds of equality checks are used in practice: In total, we observe 2,026,782 strict equality checks and 3,143,592

non-strict equality checks during the execution of all programs. To better understand to what degree coercions at equality checks influence code understandability, we measure how often a particular code location compares values of the same type (i.e., no coercion) and values of different types (i.e., coercion). "Same type" here means two types that can be trivially compared: (i) exactly the same type, (ii) in a `==` or `!=` check, any type compared to `undefined` or null, (iii) in a `===` or `!==` check, any type and `undefined`, or (iv) in a `===` or `!==` check, any non-primitive type and `undefined` or `null`.

Figure 10 gives the results for non-strict and strict comparisons. The figure shows that for almost all locations that compare values with `==` or `!=`, the values that occur at runtime are always of the same type. The results are very similar for comparisons with the strict operators `===` and `!==`.

We draw two conclusions from these results. First, developers seem to use non-strict and strict equality interchangeably. Under the assumptions that programmers fully understand the semantics of strict and non-strict equality and that they follow the advice to use strict equality when comparing values of the same type, one would expect that non-strict equality is mostly applied to different types. However, the results show that the percentage of locations that always compare the same types is almost equally high for non-strict and strict equality. Second, many non-strict equality checks could likely be refactored into strict checks. Since our study may not consider all paths, we cannot say with certainty that particular non-strict checks always compare values of the same type. Nevertheless, the results suggest that many code locations use `==` and `!=` without any need, and that these location could use `===` and `!==` instead.

5 Threats to Validity

The validity of the conclusions drawn from our results are subject to several threats. First, since our study is based on dynamic analysis, it ignores coercions triggered on paths not executed during the analysis. In particular, the results on polymorphic code locations (Figure 9a) potentially underestimate the number of coercions that occur at a location, and the results on equality checks (Figure 10) may classify a location as "always same" even though it may compare values of different types. Second, the classification of coercions into harmless and potentially harmful may be biased by the subjective experiences of the authors. We try to minimize this bias by considering informal experience reports of other JavaScript developers, e.g., in web forums and by comparing the behavior of JavaScript to other languages. Third, the subject programs of the study may not be representative for a larger population of programs. By focusing on the most popular web sites, code shared by multiple popular domains or popular third-party libraries may be overrepresented. Moreover, some of the benchmark programs contain generated JavaScript code, which may not be representative for human-written code. Fourth, the results of manually inspecting code that performs coercions are influenced by our limited ability to understand this code. To reduce this bias, we use a deobfuscation technique [18][2] to inspect minified and obfuscated code, and we interactively debug the inspected code locations. Finally, this study is limited to JavaScript, whose approach to type coercions occupies an extreme spot in the language spectrum. Our conclusions are for JavaScript and may not extend to other languages.

---

[2] `http://jsnice.org`

6 Related Work

Studying how programming languages are used in practice has a long history (for computer science standards), e.g., going back to a more than 40 years old study of Fortran programs by D. Knuth [12]. More recently, Richards et al. investigate the dynamic behavior of JavaScript programs and show that several dynamic features are widely used [21]. A study of Python programs draws similar conclusions and shows that the assumption that Python programmers only rarely use dynamic language features is false [9]. Another study [20] provides a detailed analysis of JavaScript's notorious `eval` function. In contrast to our work, none of these studies investigates type coercions. Nikiforakis et al. describe a large-scale study on how JavaScript-based web applications include code from third parties, and how these inclusions influence security [15]. Our work shares with [20] and [15] the idea to analyze in-depth how a particular language feature is used in the wild. Callau et al. describe a study of dynamic language features in Smalltalk [1]. In contrast to the above approaches and our work, they use static analysis. Their work focuses on reflection-related language features and finds that these features are used infrequently (in 1.76% of all methods).

There are various studies of how Java programmers use Java's language features. For example, Tempero et al. study the use of inheritance [24] and of fields [23]. Other work proposes an infrastructure for querying facts extracted from the source code of a corpus of Java programs, and shows how to use this infrastructure to answer various questions on how "typical" Java code is written [5]. Malayeri and Aldrich investigate whether Java programs could benefit from structural subtyping through a mixture of static and manual analysis [13]. Instead of analyzing how a feature could be used if it existed, we analyze how an already existing feature is used. To understand how much multiple dispatch is used and could be used, Muschevici et al. study programs written in six languages that support this feature and in Java, respectively [14]. All these approaches are based on static analysis of the subject programs, whereas we use dynamic analysis.

Complementary to studying source code and its execution are studies on how human subjects react to particular languages or language features. Hanenberg presents such a study on whether a static type system reduces development time [7]. Our work raises several questions that could be addressed in similar studies, e.g., how type coercions influence program understandability, or whether the conciseness of code written with coercions outweighs the potential error-proneness of coercions during development. Ocariza et al. perform studies of JavaScript errors [17] and their root causes [16]. They do not identify coercions as a particular cause of errors, which matches our finding that most coercions do not cause misbehavior.

Several approaches for inferring and checking types in JavaScript programs have been proposed, some of which raise errors on type coercions. A type system for a subset of JavaScript by Thiemann [25] reports all coercions as errors, presumably under the assumption that coercions are generally erroneous. A statically typed dialect of JavaScript, called "Dependent JavaScript", prohibits type coercions because they "often lead to subtle programming errors" [2]. Our work contradicts this assumption based on empirical evidence showing that most coercions are harmless. A type analysis by Jensen et al. warns developers about particular kinds of coercions [11]. We also classify some of them as potentially harmful, e.g., coercing `undefined` to a number. Our results could help to reduce the number of warnings of their analysis by focusing on kinds of coercions that occur rarely and that are potentially harmful. Other type inference and checking approaches for JavaScript [8, 6] do not explicitly discuss if and how they handle coercions. Furr et al. propose a profile-guided static typing approach for Ruby [4]. They report that some coercions cause type errors that forced Furr

et al. to refactor code to avoid coercions. Supporting type coercions in a static analyses is non-trivial and researchers need guidance on whether and how to support coercions. Our study provides empirical evidence that supporting type coercions is vital, along with guidance on which coercions to address first.

JavaScript's strict mode disallows some language features that are generally considered as dangerous. However, strict mode does not change the semantics of type coercions. A complementary approach to strict mode, called *restrict mode*[3], forbids several potentially harmful type coercions. At the time of this writing, the coercions allowed by restrict mode and the coercions that we classify as harmless overlap partly. For example, restrict mode does not warn about wrapped primitives in conditionals, whereas we classify them as potentially harmful, but it forbids concatenating strings and arrays, whereas we classify this operation as harmless. We believe that an empirical study of coercions provides a good base for an informed decision about which coercions to allow or disallow in a safer JavaScript subset.

7 Conclusion

This paper presents the first in-depth analysis of implicit type conversions in real-world JavaScript programs. In reference to the title of this paper, we show that most coercions are "good", few coercions are "bad", and some coercions are "ugly" but nevertheless correct. Based on the results of this study, we draw the following conclusions about real-world JavaScript programs and analyses that target them:

- Type coercions are widely used and analyses that target realistic programs should take them into account.
- Most coercions are not erroneous, and even among those kinds of coercions that seem error-prone, most coercions do not cause any misbehavior.
- The previous two conclusions lead to the third one: Research on static analyses for JavaScript faces the challenge of checking programs with various harmless coercions.
- Most coercions occur in conditions and conditional-related operations, where some value is coerced into a boolean. Static analyses that address these coercions will handle a large part of all coercions.
- A very small subset of all coercions are potentially harmful because their semantics may surprise developers. Restricted variants of JavaScript and future language designs can prohibit these coercions while still supporting most existing code.
- Most code locations that coerce types are monomorphic, and most polymorphic locations are unsurprising because they are related to conditionals that check for undefined values. That is, polymorphism caused by coercions degrades code understandability only marginally.
- Developers use non-strict and strict equality almost interchangeably. Automated refactoring approaches that soundly transform non-strict into strict equality operations seem a promising direction for future work.
- We confirm earlier results showing that the SunSpider and Octane benchmarks do not accurately represent real-world web sites [21, 19], and we extend the scope of this finding to type coercions.

Given the increasing importance of JavaScript as a language for web, mobile, desktop, and server applications, we believe that our work is an important step toward understanding how

---

[3] `http://restrictmode.org`

developers use JavaScript and toward aligning future research activities with the real-world usage of the language.

References

1   Oscar Callaú, Romain Robbes, Éric Tanter, and David Röthlisberger. How (and why) developers use the dynamic features of programming languages: the case of smalltalk. *Empirical Software Engineering*, 18(6):1156–1194, 2013.

2   Ravi Chugh, David Herman, and Ranjit Jhala. Dependent types for JavaScript. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 587–606, 2012.

3   Douglas Crockford. *JavaScript: The Good Parts*. O'Reilly, 2008.

4   Michael Furr, Jong-hoon (David) An, and Jeffrey S. Foster. Profile-guided static typing for dynamic scripting languages. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 283–300. ACM, 2009.

5   Mark Grechanik, Collin McMillan, Luca DeFerrari, Marco Comi, Stefano Crespi-Reghizzi, Denys Poshyvanyk, Chen Fu, Qing Xie, and Carlo Ghezzi. An empirical investigation into a large-scale Java open source code repository. In *Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2010.

6   Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. Typing local control and state using flow analysis. In *European Symposium on Programming (ESOP)*, pages 256–275, 2011.

7   Stefan Hanenberg. An experiment about static and dynamic type systems: doubts about the positive impact of static type systems on development time. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 22–35, 2010.

8   Phillip Heidegger and Peter Thiemann. Recency types for analyzing scripting languages. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 200–224, 2010.

9   Alex Holkner and James Harland. Evaluating the dynamic behaviour of Python applications. In *Australasian Computer Science Conference (ACSC)*, pages 17–25, 2009.

10 Ecma International. ECMAScript language specification, 5.1 edition, June 2011.

11 Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for JavaScript. In *Symposium on Static Analysis (SAS)*, pages 238–255. Springer, 2009.

12 Donald E. Knuth. An empirical study of FORTRAN programs. *Software Practice and Experience*, pages 105–133, 1971.

13 Donna Malayeri and Jonathan Aldrich. Is structural subtyping useful? An empirical study. In *European Symposium on Programming (ESOP)*, pages 95–111, 2009.

14 Radu Muschevici, Alex Potanin, Ewan D. Tempero, and James Noble. Multiple dispatch in practice. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 563–582, 2008.

15   Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter
        Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. You are what you
        include: large-scale evaluation of remote JavaScript inclusions. In *CCS*, pages 736–747,
        2012.

16   Frolin S. Ocariza Jr., Kartik Bajaj, Karthik Pattabiraman, and Ali Mesbah. An empirical
        study of client-side JavaScript bugs. In *Symposium on Empirical Software Engineering and
        Measurement (ESEM)*, pages 55–64, 2013.

17   Frolin S. Ocariza Jr., Karthik Pattabiraman, and Benjamin G. Zorn. JavaScript errors in
        the wild: An empirical study. In *International Symposium on Software Reliability Engi-
        neering (ISSRE)*, pages 100–109, 2011.

18   Veselin Raychev, Martin T. Vechev, and Andreas Krause. Predicting program properties
        from "big code". In *Principles of Programming Languages (POPL)*, pages 111–124, 2015.

19   Gregor Richards, Andreas Gal, Brendan Eich, and Jan Vitek. Automated construction of
        JavaScript benchmarks. In *Conference on Object-Oriented Programming, Systems, Lan-
        guages, and Applications (OOPSLA)*, pages 677–694, 2011.

20   Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. The eval that men do -
        a large-scale study of the use of eval in JavaScript applications. In *European Conference
        on Object-Oriented Programming (ECOOP)*, pages 52–78, 2011.

21   Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An analysis of the dynamic
        behavior of JavaScript programs. In *Conference on Programming Language Design and
        Implementation (PLDI)*, pages 1–12, 2010.

22   Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. Jalangi: A selec-
        tive record-replay and dynamic analysis framework for JavaScript. In *European Software
        Engineering Conference and Symposium on the Foundations of Software Engineering (ES-
        EC/FSE)*, pages 488–498, 2013.

23   Ewan D. Tempero. How fields are used in Java: An empirical study. In *Australian Software
        Engineering Conference (ASWEC)*, pages 91–100, 2009.

24   Ewan D. Tempero, James Noble, and Hayden Melton. How do Java programs use inher-
        itance? An empirical study of inheritance in Java software. In *European Conference on
        Object-Oriented Programming (ECOOP)*, pages 667–691. Springer, 2008.

25   Peter Thiemann. Towards a type system for analyzing JavaScript programs. In *European
        Symposium on Programming (ESOP)*, pages 408–422, 2005.