Adaptive Context-sensitive Analysis for JavaScript

Shiyi Wei and Barbara G. Ryder

Department of Computer Science
Virginia Tech
Blacksburg, VA, USA
{wei, ryder}@cs.vt.edu

Abstract

Context sensitivity is a technique to improve program analysis precision by distinguishing between function calls. A specific context-sensitive analysis is usually designed to accommodate the programming paradigm of a particular programming language. JavaScript features both the object-oriented and functional programming paradigms. Our empirical study suggests that there is no single context-sensitive analysis that always produces precise results for JavaScript applications. This observation motivated us to design an adaptive analysis, selecting a context-sensitive analysis from multiple choices for each function. Our two-staged adaptive context-sensitive analysis first extracts function characteristics from an inexpensive points-to analysis and then chooses a specialized context-sensitive analysis per function based on the heuristics. The experimental results show that our adaptive analysis achieved more precise results than any single context-sensitive analysis for several JavaScript programs in the benchmarks.

1998 ACM Subject Classification D.3.4 Processors, F.3.2 Semantics of Programming Languages, D.2.5 Testing and Debugging

Keywords and phrases Context Sensitivity, JavaScript, Static Program Analysis

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2015.712

1 Introduction

JavaScript is the *lingua franca* of client-side web applications and is becoming the most popular programming language choice for open source projects [13]. Its flexible programming paradigm is one of the reasons behind its popularity. JavaScript is known as a dynamic object-oriented language, supporting prototype-based programming [9, 20], a different model from class-based languages. However, it also supports features of functional programming (e.g., first class functions). This flexibility helps in programming JavaScript applications for different requirements and goals, but it also renders program analysis techniques more complicated and often ineffective.

Context sensitivity is a general technique to achieve more precise program analysis by distinguishing between calls to a specific function. Historically, call-strings and functional are the two approaches to enable context sensitivity in an analysis [14]. The call-strings approach, using the information on the call stack as a context element[1], originally was explored by Shivers (i.e., known as call-site sensitivity or k-CFA) for functional programming languages [15] and was later adapted to object-oriented languages such as Java [2]. The functional approach, using information about the computation state as a context element, was investigated in several variations for object-oriented languages (e.g., [1, 10]). Several

[1] A *context element* refers to the label that is used to distinguish between different function calling contexts.

29th European Conference on Object-Oriented Programming (ECOOP'15).
Editor: John Tang Boyland; pp. 712–734

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

studies were performed to compare the precision of different context-sensitive analyses for Java [7, 16]. These studies revealed that object sensitivity [10], an functional approach that uses the receiver object represented by its allocation site as a context element, often produced more precise results than call-site sensitivity in Java applications.

Despite the challenges of effective analysis for JavaScript programs, different context-sensitive analyses have been developed to improve precision. Jensen *et al.* implemented object sensitivity in the TAJS analysis framework [4]. Several other context-sensitive analyses were designed for specific features in JavaScript programs (e.g., dynamic property accesses [19] and object property changes [22]). These approaches have been demonstrated effective when certain program features are present. Recently, Kashyap *et al.* conducted a study to compare the performance and precision of different context-sensitive analyses for JavaScript [5]. Unlike the results for Java analyses [7, 16], the authors concluded that there was no clear winner context-sensitive analysis for JavaScript across all benchmarks [5].

Because JavaScript features flexible programming paradigms and no single context-sensitive analysis seems best for analyzing JavaScript programs, there are opportunities for an adaptive (i.e., multi-choice) analysis to improve precision. In this work, we first performed a fine-grained study that compares the precision of four different JavaScript analyses on the function level (Section 2.2) on the same benchmarks used by Kashyap *et al.* [5]. We observed that JavaScript functions in the same program may benefit from use of different context-sensitive analyses depending on specific characteristics of the functions.

The results of our empirical study guided us to design a novel adaptive context-sensitive analysis for JavaScript that selectively applies a specialized context-sensitive analysis per function chosen from call-site, object and parameter sensitivity. This two-staged analysis first applied an inexpensive points-to analysis (i.e., mostly context-insensitive, Section 2.2) to a JavaScript program to extract function characteristics. Each function characteristic is relevant to the precision of a context-sensitive analysis of the function (e.g., the call sites that invoke function *foo* determine the context elements to be generated if call-site sensitivity is applied to *foo*). We designed heuristics according to our observations on the relationship between function characteristics and the precision of a specific analysis using the empirical results on the benchmark programs. Finally, an adaptive analysis based on the heuristics-based selection of a context-sensitive analysis per function was performed. We have evaluated our new approach on two sets of benchmarks. The experimental results show that our adaptive context-sensitive analysis produces more precise results than any single context-sensitive analysis we evaluated for several JavaScript programs and that the heuristics for selecting a context-sensitive analysis per function are fairly accurate.

The major contributions of this work are:

- *A new empirical study on JavaScript benchmarks to compare the precision of different context-sensitive analyses* (i.e., 1-call-site, 1-object and 1st-parameter as defined in Section 2.1). We measure the precision *per function* on two simple clients of points-to analysis (i.e., *Pts-Size* and *REF*). We have made several observations: (i) any specific context-sensitive analysis is precise only for a subset of programs in the benchmarks. More interestingly, any specific context-sensitive analysis is often effective only on a portion of a program. (ii) The precision of a context-sensitive analysis also is dependent on the analysis client. These findings motivated and guided us to design a novel JavaScript analysis.
- *An adaptive context-sensitive analysis for JavaScript.* The heuristics to select from various context-sensitive analyses (i.e., 1-call-site, 1-object and ith-parameter) for a function are based on the function characteristics computed from an inexpensive points-to solution. This adaptive analysis is the *first* analysis for JavaScript that automatically and selectively

applies a context-sensitive analysis depending on the programming paradigms (i.e., coding styles) of a function.

- *An empirical evaluation of the adaptive context-sensitive analysis on two sets of benchmark programs.* The experimental results show that our adaptive analysis was more precise than any single context-sensitive analysis for several applications in the benchmarks, especially for those using multiple programming paradigms. Our results also show that the heuristics were accurate for selecting appropriate context-sensitive analysis on the function level.

Overview. Section 2 introduces various context-sensitive analyses and then presents our motivating empirical study. Section 3 discusses heuristics that represent the relationship between function characteristics and analysis precision. Section 4 describes the design of our new analysis. Section 5 presents experimental results. Section 6 further discusses related work. Section 7 offers conclusions.

2 Background and Empirical Study

In this section, we first introduce the context-sensitive analyses used in our study. We then present the new comparisons among different context-sensitive analyses with empirical results and observations.

2.1 Context Sensitivity

As discussed in Section 1, various context-sensitive analyses have been designed for different programming languages. In this section, we only discuss the approaches that are most relevant to our empirical study of context-sensitive analysis for JavaScript. We provide more discussions on related context-sensitive analyses in Section 6.

Call-strings approach. A call-strings approach distinguishes function calls using information on the call stack. The most widely known call-strings approach is call-site-sensitive (k-CFA) analysis [15]. A k-call-site sensitive analysis uses a sequence of the top k call sites on the call stack as the context element. k is a parameter that determines the maximum length of the call string maintained to adjust the precision and performance of call-site-sensitive analysis. We used *1-call-site sensitivity* for comparison. 1-call-site-sensitive analysis separately analyzes each different call site of a function. Intuitively in the code example below, 1-call-site-sensitive analysis will analyze function *foo* in two calling contexts *L1* and *L2*, such that local variables (including parameters) of *foo* will be analyzed independently for each context element.

```
L1: x.foo(p1, p3);
L2: y.foo(p2, p4);
```

Functional approach. A functional approach distinguishes function calls using information about the computation state at the call. Object sensitivity analyzes a function separately for each of the abstract object names on which this function may be invoked [10]. Milanova *et al.* presented object sensitivity as a parameterized k-object-sensitive analysis, where k denotes the maximum sequence of allocation sites to represent an object name. We used *1-object sensitivity* for comparison. 1-object-sensitive analysis separately analyzes a function for each of its receiver objects with a different allocation site. Intuitively in the code example above, 1-object-sensitive analysis will analyze function *foo* separately if x and/or y may point to

different abstract objects. If x points to objects O_1 and O_2, while y points to object O_3, 1-object-sensitive analysis will analyze function *foo* for three context elements differentiated as O_1, O_2 and O_3.

Other functional approaches presented use the computation state of the parameter instead of the receiver object as a context element [1, 19]. The Cartesian Product Algorithm (CPA) uses tuples of parameter types as a context element for Self [1]. The context-sensitive analysis presented by Sridharan *et al.*, designed specifically for JavaScript programs, analyzes a function separately using the values of a parameter p if p is used as the property name in a dynamic property access (e.g., *v[p]*) [19]. To capture these approaches, we define a simplified, parameterized ith-parameter-sensitive analysis, where i means we use the abstract objects corresponding to the *ith* parameter as a context element. We used *1st-parameter sensitivity* for comparisons. Intuitively in the code example above, 1st-parameter-sensitive analysis will analyze function *foo* separately if *p1* and/or *p2* may point to different abstract objects. If *p1* points to object O_4, while *p2* points to object O_4 and O_5, 1st-parameter-sensitive analysis will analyze function *foo* for two context elements distinguished as O_4 and O_5.

2.2 Empirical Study

There is no theoretical comparison between the functional and call-strings approaches to context sensitivity that proves one better than the other. Therefore, we study the precision of different context-sensitive analyses in practice based on experimental observations. Such comparisons have been conducted for call-site and object sensitivity on Java [7, 16] as well as JavaScript [5] applications. Object sensitivity produced more precise results for Java benchmarks [7, 16], while there was no clear winner across all benchmarks for JavaScript [5]. The latter observation motivated us to perform an in-depth, fine-grained, function-level study which led to our design of a new context-sensitive analysis for JavaScript.

Hypothesis. Our hypothesis is that a specific context-sensitive analysis may be more effective on a portion of a JavaScript program (i.e., some functions), while another kind of context sensitivity may produce better results for other portions of the same program. To test this hypothesis, we compared the precision of different context-sensitive analyses at the function level (i.e., we collect the results of different analyses for a specific function and compare their precision). In contrast, all previous work [7, 16, 5] reported overall precision results per benchmark program. The results of our empirical study were obtained on a 2.4 GHz Intel Core i5 MacBook Pro with 8GB memory running the Mac OS X 10.10 operating system.

Analyses for comparisons. We compared across four different flow-insensitive analyses to study their precision. The *baseline* analysis is an analysis that applies the default context-sensitive analysis for JavaScript in *WALA*[2] (i.e., only uses 1-call-site-sensitive analysis for the constructors to name abstract objects by their allocation sites and for nested functions to properly access variables accessible through lexical scoping [19]). In the implementation, the other three analyses (i.e., 1-call-site, 1-object and 1st-parameter) all apply this default analysis. In principle, these analyses should be at least as precise as the baseline analysis for all functions.

[2] Our implementation is based on the IBM T.J. Watson Libraries for Analysis (*WALA*). `http://wala.sourceforge.net/`

Analysis clients and precision metrics. We compare precision results on two simple clients of points-to analysis. Points-to analysis calculates the set of values a reference property or variable may have during execution, an important enabling analysis for many clients. The first client, *Pts-Size*, is a points-to query returning the cardinality of the set of all values of all local variables in a function (i.e., the total number of abstract objects pointed to by local variables). The second client, *REF* [22], is a points-to query returning the cardinality of the set of all property values in all the property read (e.g., x=$y.p$) or call statements (e.g., $x = y.p(...)$) in a function (i.e., the total number of abstract objects returned by all the property lookups). Because both clients count the number of abstract objects returned and object-naming scheme is unified across different analyses, if an analysis A_1 produces a smaller result than another analysis A_2 for a function *foo*, we say that A_1 is more precise than A_2 for *foo*.

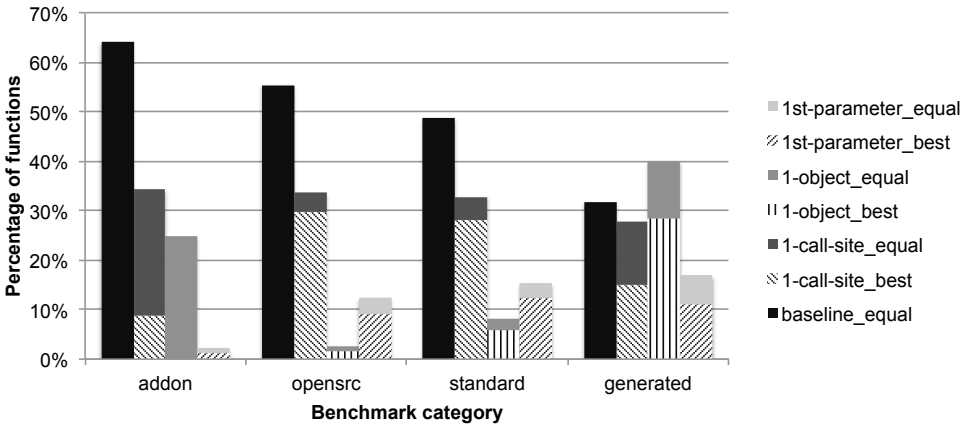
Benchmarks. We conduct our comparisons on the same set of benchmarks used for the study performed by Kashyap *et al.* [5]. There are in total 28 JavaScript programs divided into four categories: *standard* (i.e., from SunSpider[3] and V8[4]), *addon* (i.e., Firefox browser plugins), *generated* (i.e., from the Emscripten LLVM test suite[5]) and *opensrc* (i.e., open source JavaScript frameworks). There are seven programs in each benchmark category.[6] In our study, all benchmark programs are transformed with function extraction for correlated property accesses [19] before running any analysis. This program transformation preserves the semantics of the original programs and may help handle dynamic property accesses more accurately [19].

Results. We ran each analysis of a benchmark program under a time limit of 10 minutes. The baseline and 1-call-site-sensitive analyses finished analyzing all 28 programs under the time limit. 1-object-sensitive analysis timed out on 4 programs (i.e., *linq_aggregate*, *linq_enumerable* and *linq_functional* in the *opensrc* benchmarks and *fourinarow* in the *generated* benchmarks), while 1st-parameter-sensitive analysis timed out on 2 programs (i.e., *fasta* and *fourinarow* in the *generated* benchmarks).

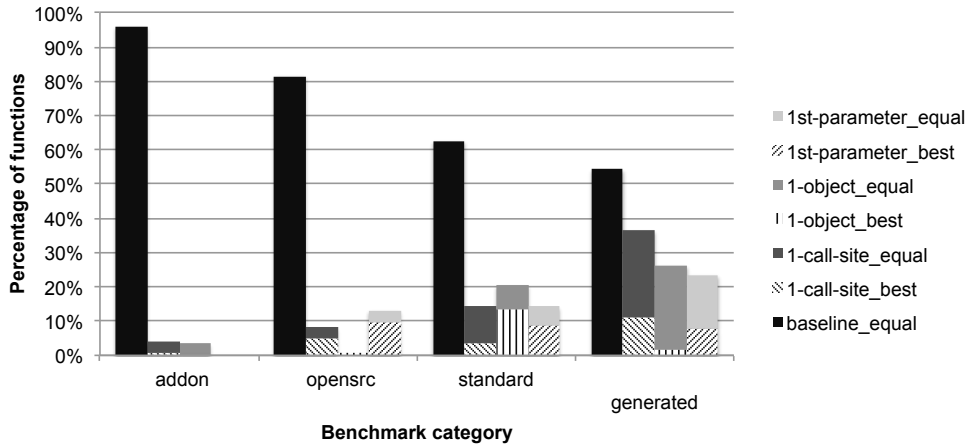
Figures 1a and 1b show the relative precision results for *Pts-Size* and *REF*, respectively. In both figures, the horizontal axis represents the results from four benchmark categories (i.e., *standard*, *addon*, *generated* and *opensrc*) and the vertical axis represents the percentages of functions in each benchmark category on which an analysis produces the *best* results (i.e., more precise results than those from all other three analyses) or *equally precise* results. We consider the results of an analysis as *equally precise* as follows. (i) Baseline analysis is *equally precise* on a function if its results are as precise as each of the other three context-sensitive analyses. (ii) 1-call-site-sensitive, 1-object-sensitive or 1st-parameter-sensitive analysis is *equally precise* on a function if the results are more precise results than baseline analysis, and if the analysis does not produce the *best* results but the results are at least as precise as the other two context-sensitive analyses. This definition indicates that multiple context-sensitive analyses (e.g., 1-call-site-sensitive and 1-object-sensitive analyses) may produce *equally precise* results on a function.

For example, the left four bars in Figure 1a present the precision results for the *addon*

[3] `https://www.webkit.org/perf/sunspider/sunspider.html`
[4] `http://v8.googlecode.com/svn/data/benchmarks/v7/run.html`
[5] `http://kripken.github.io/emscripten-site/`
[6] Details of the benchmark programs were provided in Kashyap *et al.* [5].



(a) Relative precision results for the *Pts-Size* client.



(b) Relative precision results for the *REF* client.

Figure 1 Comparison results between baseline, 1-call-site-sensitive, 1-object-sensitive and 1st-parameter-sensitive analyses. Bars show the percentage of functions on which an analysis is *best* or *equally precise*.

benchmarks for the *Pts-Size* client. The *baseline_equal* bar (i.e., the left most) shows that baseline analysis achieved as precise results as those from all three other analyses for 64% of the functions in the *addon* benchmarks, indicating context sensitivity does not make much difference for more than two thirds of the functions in these programs for the *Pts-Size* client. The *1-call-site_best* and *1st-parameter_best* bars (i.e., the parts of the second and fourth bars from left filled with patterns) show that 1-call-site-sensitive and 1st-parameter-sensitive analyses produced more precise results than all other analyses for 9% and 1.5% of the functions in the *addon* benchmarks, respectively. The *1-object_best* result missing from the third bar from left indicates that 1-object-sensitive analysis failed to produce the most precise results for any function in *addon* benchmarks. Nevertheless, the *1-object_equal* bar shows 1-object-sensitive analysis achieved *equally precise* results with 1-call-site-sensitive and/or 1st-parameter-sensitive analyses for 25% of the functions in the *addon* benchmarks.

Comparing with the *1-call-site_equal* (26%) and *1st-parameter_equal* (1%) bars, we can predict that 1-call-site-sensitive and 1-object-sensitive analyses had similar precision on a quarter of the functions in the *addon* benchmarks.

The *baseline_equal* bars in Figure 1a show that analysis of a large percentage of functions in the benchmarks does not benefit from context sensitivity in terms of the *Pts-Size* results (i.e., from 32% for the *generated* benchmarks to *64%* for the *addon* benchmarks). Also, 1-call-site-sensitive analysis had relatively consistent impact in the benchmarks, achieving *best* or *equally precise* results for around 30% functions across all benchmark categories. In contrast, the precision of 1-object-sensitive analysis results seems dependent on the benchmark. Having little impact on the precision of the *opensrc* benchmarks, 1-object-sensitive analysis produced *best* results for 29% of the functions in the *generated* benchmarks with an additional 11% of the functions achieving *equally best* results. 1st-parameter-sensitive analysis, less studied in previous work, produced *best* results for about 10% functions in the *opensrc*, *standard* and *generated* benchmarks, a reasonable technique to improve precision for these programs. It is also interesting to learn from Figure 1a that different context-sensitive analyses may produce *equally precise* results on many functions in some benchmark categories. 1-call-site-sensitive and 1-object-sensitive analyses produced *equally best* results for 25% functions in the *addon* benchmarks. 1-call-site-sensitive, 1-object-sensitive and 1st-parameter-sensitive analyses produced *equally best* results for 3% functions in the *generated* benchmarks.

Recall that the *REF* client uses data from different parts of the points-to results than the *Pts-Size* client; in addition, the *REF* client may query the points-to results (i.e., all the property lookup statements in a function) less frequently than the *Pts-Size* client (i.e., all local variables in a function). Overall in Figure 1b, context sensitivity improves precision less over baseline analysis for the *REF* than for the *Pts-Size* client. Baseline analysis produced as precise results as all other three analysis for more than 50% of the functions in all benchmark categories. About 96% of the functions in the *addon* benchmarks did not benefit from any context-sensitive analysis over the baseline analysis. 1-call-site-sensitive analysis achieved dramatically better results for the *Pts-Size* client than *REF* client in *addon*, *opensrc* and *standard* benchmarks. 1-object-sensitive analysis also achieved much better results for the *Pts-Size* client than the *REF* client in the *generated* benchmarks. On the other hand, 1st-parameter-sensitive analysis still remains effective in the *opensrc*, *standard* and *generated* benchmarks.

Summary. First, the effectiveness of specific context-sensitive analysis for JavaScript functions is sensitive to the coding style. For example, 1-call-site-sensitive, 1-object-sensitive and 1st-parameter-senstive analyses each produced *best* results on a large percentage of functions in the programs from the *generated* benchmarks. Second, the precision of context-sensitive analysis also depends on the analysis client.

Based on these observations, we believe JavaScript programs can benefit from an adaptive context-sensitive analysis that chooses an appropriate context-sensitive analysis for a specific function. We used these observations as guidance to design our new analysis.

3 Function Characteristics and Heuristics

A context-sensitive analysis is designed to be useful for a specific programming paradigm. For example, object-sensitive analysis targets the class-based model of object-oriented languages. The results from Section 2 indicate that JavaScript functions in one program may benefit from different context-sensitive analyses depending on the coding style of the functions. In

Table 1 Function characteristics.

	1-call-site	1-object	1st-parameter
context element approximations	*FC1: CSNum*	*FC4: RCNum*	*FC6: 1ParNum*
	FC2: EquivCSNum		
client-related metrics	*FC3: AllUse*	*FC5: ThisUse*	*FC7: 1ParName*
			FC8: 1ParOther

this section, we present the function characteristics we extracted from the baseline analysis results and then investigate heuristics that represent the relations between these function characteristics and the precision of context-sensitive analyses. Finally, we use these heuristics to select the appropriate context-sensitive analysis per function in our adaptive algorithm (Section 4).

3.1 Function Characteristics

For a JavaScript function, we extracted characteristics from the baseline analysis results that are relevant to the precision of context-sensitive analyses for a specific client. The goal is to extract function characteristics that (i) intuitively are relevant to the precision of a specific analysis for a particular client, and (ii) do not require more costly analysis than a baseline points-to analysis. Table 1 shows that for a JavaScript function *foo*, we extract eight function characteristics (i.e., *FC1*, *FC2*, ..., *FC8*). Each function characteristic is related to the precision of a specific context-sensitive analysis (i.e., *FC1-FC3*, *FC4-FC5*, and *FC6-FC8* are related to the precision of 1-call-site, 1-object and 1st-parameter, respectively).

For a specific context-sensitive analysis, we extracted two kinds of function characteristics: *context element approximations* and *client-related metrics*. A context element approximation predicts the number of distinct context elements generated for a function by a context-sensitive analysis, which determines its ability to distinguish between function calls. A client-related metric predicts the effectiveness of a context-sensitive analysis on a JavaScript function for a particular client. *FC1-FC2*, *FC4* and *FC6* in Table 1 are the context element approximations we designed for 1-call-site-sensitive, 1-object-sensitive and 1st-parameter-sensitive analyses, respectively. For example, *FC4-RCNum* presents an approximation of the number of receiver objects on which a function is called, computed from the baseline points-to results. *FC3*, *FC5* and *FC7-FC8* are the client-related metrics we designed for the *Pts-Size* client[7] for 1-call-site, 1-object and 1st-parameter sensitivity, respectively. For example, *FC5-ThisUse* measures the usage frequency of the *this* object in the function body, because frequent use of the *this* object indicates that the precision of the *Pts-Size* client on the function depends on how accurately the *this* object is analyzed. Because 1-object-sensitive analysis potentially analyzes the *this* object more precisely, we use *FC5-ThisUse* to predict the effectiveness of 1-object-sensitive analysis on a function for the *Pts-Size* client. We now will define each function characteristic.

[7] In this work, we use the *Pts-Size* client to demonstrate the effectiveness of our approach because the empirical results in Section 2.2 suggest that the *Pts-Size* client is relatively more sensitive to the choice of context-sensitive analyses.

Function characteristics for 1-call-site-sensitive analysis. Recall that a 1-call-site-sensitive analysis uses the immediate call site of a function as the context element. We define *FC1*, the *CSNum* metric, as follows:

- *FC1-CSNum*: for function *foo*, the number of call sites that invoke *foo* in the baseline call graph *G*.

Although *FC1* approximates the number of context elements that a 1-call-site-sensitive analysis would generate for *foo*, this metric may not be directly relevant to the precision of 1-call-site-sensitive analysis. Intuitively, if function *foo* is invoked from two call sites *CS1* and *CS2*, the analysis precision on *foo* is not likely to benefit from distinguishing between these two call sites if the parameters of *CS1* and *CS2* have the same values because these parameters are used in *foo* as local variables. More precisely, we define two call sites, *CS1: p0.foo(p1, p2, ... , pn)* and *CS2: p0'.foo(p1', p2', ..., pn')*, to be *equivalent* if for each pair of receiver objects and parameters (i.e., *pi* and *pi'*) in *CS1* and *CS2*, the points-to sets of *pi* and *pi'* are the same. We then define *FC2*, the *EquivCSNum* metric using this definition of equivalent call sites, as follows:

- *FC2-EquivCSNum*: for function *foo*, the number of equivalence classes of call sites that invoke *foo* in the baseline call graph *G*.

Recall that the *Pts-Size* client calculates the cardinality of the set of abstract objects to which a local variable of *foo* may point. Intuitively, the precision of the *Pts-Size* client depends on the receiver object or parameters that are frequently used as local variables in the function body. For example, if a parameter *p* is never used in *foo*, even if 1-call-site-sensitive analysis distinguishes call sites that pass different values of *p*, the results of the *Pts-Size* client may not be different because *p* is never used locally. Theoretically, 1-call-site sensitivity may distinguish objects passed through any parameter as well as receiver objects via call sites. We define *FC3*, the *AllUse* metric, as follows:

- *FC3-AllUse*: for function *foo*, the total number of uses of the *this* object and all parameters.

Function characteristics for 1-object-sensitive analysis. Recall that 1-object-sensitive analysis distinguishes calls to a function if they correspond to different receiver objects. To approximate the number of context elements generated by 1-object-sensitive analysis for function *foo*, we define *FC4*, the *RCNum* metric, as follows:

- *FC4-RCNum*: for function *foo*, the total number of abstract receiver objects from all call sites that invoke *foo* in the baseline call graph *G*.

Naturally, 1-object-sensitive analysis would be effective on functions implemented with the object-oriented programming paradigm. The behavior of these functions is dependent on the objects on which they are called. Uses of the *this* object in a function is common in the object-oriented programming paradigm and 1-object-sensitive analysis should produce relatively precise results. We define *FC5*, the *ThisUse* metric, as follows:

- *FC5-ThisUse*: for function *foo*, the total number of uses of the *this* object.

Function characteristics for 1st-parameter-sensitive analysis. The ith-parameter sensitivity is designed to be effective when a specific parameter (e.g., the first parameter for 1st-parameter-sensitive analysis) has large impact on analysis precision. 1st-parameter-sensitive analysis uses the objects that the 1st parameter points to as context elements. We define *FC6*, the *1ParNum* metric, as follows:

- *FC6-1ParNum*: for function *foo*, the total number of abstract objects to which the 1st parameter may point from all call sites that invoke *foo* in the baseline call graph *G*.

If a parameter p is frequently used in a function, it may be more important to apply context-sensitive analysis on p than on the receiver object. Also, if p is used as a property name in dynamic property accesses, using context sensitivity to distinguish the values of p significantly improves analysis precision [19]. We define *FC7*, the *1ParName* metric, and *FC8*, the *1ParOther* metric, as follows:

- *FC7-1ParName*: for function *foo*, the total number of uses of the 1st parameter as a property name in dynamic property accesses.
- *FC8-1ParOther.* for function *foo*, the total number of uses of the 1st parameter not as a property name.

3.2 Heuristics

The function characteristics defined in Section 3.1 are intuitive and easy to calculate from the baseline points-to graph and call graph. Nevertheless, it is still not clear how these function characteristics are related to the precision of a context-sensitive analysis. In this section, we use empirical data to design the heuristics that define the relations between function characteristics and analysis precision.

Our goal is to select an appropriate analysis for a function given the set of its function characteristics. The heuristics are not obvious given that there are multiple context-sensitive analysis choices. To design useful heuristics, we first compared the precision of a pair of analyses on the function level and observed the impact of a subset of function characteristics on these two analyses. We then applied these heuristics to adaptively choose an appropriate context-sensitive analysis using the function characteristics (Section 4). More specifically, for the *Pts-Size* results from the benchmarks (Section 2.2), we compared the precision between all 2-combinations of baseline, 1-call-site-sensitive, 1-object-sensitive and 1st-parameter-sensitive analyses and derived the heuristics to select an analysis from each of the combinations.

For example, to choose between the baseline analysis and 1-call-site-sensitive analysis, we obtained the relevant subset of function characteristics (i.e., *FC1-FC3*) and the *Pts-Size* results of baseline and 1-call-site-sensitive analyses for each function *foo* in the benchmarks. If the *Pts-Size* result from 1-call-site-sensitive analysis is more precise than baseline analysis, 1-call-site sensitivity should be chosen when analyzing *foo*; otherwise, baseline analysis should be chosen. Given the list of function characteristics and corresponding analysis choices on the benchmark functions, we first used a machine learning algorithm[8] to get the relationship (i.e., presented as a decision tree) between the function characteristics and analysis choice. We then manually adjusted the initial decision tree based on domain knowledge to decide on the heuristic, in order to ensure that the heuristic is intuitive and easy to interpret while the classifications still maintain good accuracy.

We report the accuracy of an analysis choice (e.g., 1-call-site-sensitive analysis) using the standard information retrieval metrics of *precision* and *recall*. Assuming *S1* is the set of all functions in the benchmarks where 1-call-site-sensitive analysis produces more precise results than baseline analysis and *S2* is the set of functions where 1-call-site-sensitive analysis is chosen by the heuristic. The precision of 1-call-site sensitivity classification is computed as $P_{1-call-site} = \frac{|S1 \bigcap S2|}{|S2|}$ and the recall is computed as $R_{1-call-site} = \frac{|S1 \bigcap S2|}{|S1|}$. The *balanced F-score*, the harmonic mean of the precision and recall, is computed as $F_{1-call-site}$

[8] We used the C4.5 classifier [12] implemented in Weka data mining software (`http://www.cs.waikato.ac.nz/ml/weka/`) to derive the initial decision tree.

$= 2 \times \frac{P_{1-call-site} \times R_{1-call-site}}{P_{1-call-site} + R_{1-call-site}}$, where $F_{1-call-site}$ has its best value at 1 and worst score at 0 for choosing 1-call-site-sensitive analysis by this heuristic.

Figure 2 shows the heuristics to make a choice between each pair of the analyses using function characteristic values. We discuss each pair of the analyses in turn:

Baseline vs. 1-call-site. Three function characteristics (i.e., *FC1-FC3*) are relevant to the precision of 1-call-site-sensitive analysis for the *Pts-Size* client. For 1-call-site-sensitive analysis to produce more precise results than baseline analysis, the *prerequisite* is that there is more than one distinct 1-call-site-sensitive context element (i.e., *FC1 > 1*). Figure 2a shows the heuristic to choose between baseline and 1-call-site-sensitive analyses. 1-call-site-sensitive analysis is chosen over baseline analysis for a function *foo* if there is more than one equivalence class of call sites that invoke *foo* (i.e., *FC2 > 1*). This result indicates that the effectiveness of 1-call-site sensitivity depends on its ability to distinguish call sites with different receiver objects or different corresponding parameters. The balanced F-scores for baseline and 1-call-site-sensitive analyses in this heuristic are 0.46 and 0.8, respectively.

Baseline vs. 1-object. *FC4* and *FC5* are relevant to the precision of 1-object-sensitive analysis for the *Pts-Size* client. For 1-object-sensitive analysis to produce more precise results than baseline analysis, the *prerequisite* is that there is more than one 1-object-sensitive context element (i.e., *FC4 > 1*). Figure 2b shows the heuristic to choose between baseline and 1-object-sensitive analyses. 1-object-sensitive analysis is chosen over baseline analysis for a function *foo* if the *this* object is used at least once in the function body of *foo* (i.e., *FC5 > 0*). This result suggests that 1-object-sensitive analysis is useful in terms of *Pts-Size* client for a function *foo* whose behavior relies on the values of the *this* object, even for a small number of 1-object-sensitive context elements for *foo*. The balanced F-scores for baseline and 1-object-sensitive analyses in this heuristic are 0.65 and 0.79, respectively.

Baseline vs. 1st-parameter. Three function characteristics (i.e., *FC6-FC8*) are relevant to the precision of 1st-parameter-sensitive analysis for the *Pts-Size* client. For 1st-parameter-sensitive analysis to produce more precise results than baseline analysis, the *prerequisite* is that there is more than one 1st-parameter-sensitive context element (i.e., *FC6 > 1*). Figure 2c shows the heuristic to choose between baseline and 1st-parameter-sensitive analyses. 1st-parameter-sensitive analysis is chosen over baseline analysis for a function *foo* if the first parameter of *foo* is used (i.e., as the property name in dynamic property accesses or otherwise) at least once in the function body of *foo* (i.e., *FC7 > 0 OR FC8 > 0*). Similar to 1-object sensitivity, 1st-parameter sensitivity is another functional approach that distinguishes calls based on the computation states of a parameter. It is expected for 1-object-sensitive or 1st-parameter-sensitive analysis to be effective on the function *foo* if the values of the *this* object or the first parameter affect the behavior of *foo*. The balanced F-scores for baseline and 1st-parameter-sensitive analyses in this heuristic are 0.49 and 0.83, respectively.

1-call-site vs. 1-object. To select between 1-call-site-sensitive and 1-object-sensitive analyses, function characteristics related to both are considered (i.e., *FC1-FC5*). In our adaptive analysis, two context-sensitive analyses are compared for a function when both of them would be chosen over baseline analysis (see Section 4). As a consequence, the *prerequisite* for the heuristic in this case is the number of equivalence classes of call sites is larger than 1 (i.e., *FC2 > 1*) and the *this* object is used at least once (i.e., *FC5 > 0*). Figure 2d shows the heuristic to choose between 1-call-site-sensitive and 1-object-sensitive analyses. The heuristic

```
FC2-EquivCSNum = 1: baseline
FC2-EquivCSNum > 1: 1-call-site
```

(a) Baseline vs. 1-call-site.

```
FC5-ThisUse = 0: baseline
FC5-ThisUse > 0: 1-object
```

(b) Baseline vs. 1-object.

```
FC7-1ParName = 0 AND FC8-1ParOther = 0: baseline
FC7-1ParName > 0 OR FC8-1ParOther > 0: 1st-parameter
```

(c) Baseline vs. 1st-parameter.

```
FC4-RCNum / FC2-EquivCSNum <= 0.8: 1-call-site
FC4-RCNum / FC2-EquivCSNum > 0.8
|   FC5-ThisUse / FC3-AllUse <= 0.375: 1-call-site
|   FC5-ThisUse / FC3-AllUse > 0.375: 1-object
```

(d) 1-call-site vs. 1-object.

```
FC7-1ParName = 0
|   FC8-1ParOther / FC3-AllUse <= 0.19: 1-call-site
|   FC8-1ParOther / FC3-AllUse > 0.19
|   |   FC6-1ParNum / FC2-EquivCSNum <= 3.8
|   |   |   FC8-1ParOther / FC3-AllUse <= 0.35: 1-call-site
|   |   |   FC8-1ParOther / FC3-AllUse > 0.35: 1st-parameter
|   |   FC6-1ParNum / FC2-EquivCSNum > 3.8: 1st-parameter
FC7-1ParName > 0: 1st-parameter
```

(e) 1-call-site vs. 1st-parameter.

```
FC7-1ParName = 0
|   FC5-ThisUse / FC8-1ParOther <= 0.8: 1st-parameter
|   FC5-ThisUse / FC8-1ParOther > 0.8
|   |   FC5-ThisUse / FC8-1ParOther <= 1.34
|   |   |   FC4-RCNum / FC6-1ParNum < 0.5: 1st-parameter
|   |   |   FC4-RCNum / FC6-1ParNum >= 0.5
|   |   |   |   FC4-RCNum / FC6-1ParNum <= 1: unknown
|   |   |   |   FC4-RCNum / FC6-1ParNum > 1: 1-object
|   |   FC5-ThisUse / FC8-1ParOther > 1.34: 1-object
FC7-1ParName > 0: 1st-parameter
```

(f) 1-object vs. 1st-parameter.

Figure 2 Heuristics to select between a pair of analyses.

consists of the relationship between the metrics of both analyses. 1-call-site-sensitive analysis is selected if it generates a greater number of context elements than 1-object-sensitive analysis (i.e., *FC4 / FC2 <= 0.8*) for a function. This result suggests that 1-call-site-sensitive and 1-object-sensitive analyses in this case are empirically comparable in terms of precision. The relationship between the numbers of context elements generated by each analysis on *foo* indicates which context-sensitive analysis may be more precise for that function. When the

number of receiver objects that invoke *foo* is close to or larger than the number of equivalence classes of call sites (i.e., *FC4 /FC2 > 0.8*), if the *this* object is used quite frequently (i.e., *FC5 / FC3 > 0.375*) in *foo*, 1-object-sensitive analysis is more precise for *foo*; otherwise (i.e., *FC5 / FC3 <= 0.375*), 1-call-site-sensitive analysis is selected. This result is intuitive in that 1-object-sensitive analysis produces more precise results than 1-call-site-sensitive analysis for the *Pts-Size* client when (i) 1-object-sensitive analysis generates a number of context elements and (ii) the behavior of a function is heavily dependent on the values of the receiver object. The balanced F-scores for 1-call-site-sensitive and 1-object-sensitive analyses in this heuristic are 0.67 and 0.8, respectively.

1-call-site vs. 1st-parameter. Function characteristics *FC1-FC3* and *FC6-FC8* are considered to select between 1-call-site-sensitive and 1-object-sensitive analyses. The *prerequisite* for this comparison is *FC2 > 1* and the first parameter of the function is used at least once (i.e., *FC7 > 0 OR FC8 > 0*). Figure 2e shows the heuristic to choose between 1-call-site-sensitive and 1-object-sensitive analyses. 1st-parameter-sensitive analysis is always selected if the first parameter is ever used as a property name in dynamic property accesses because the dynamic property accesses in JavaScript make analysis results very imprecise [19] and 1st-parameter sensitivity is a technique that significantly improves the analysis precision in this situation. In other cases, 1-call-site sensitive analysis is preferred if uses of the first parameter are not important to the function behavior (i.e., *FC8 / FC3 <= 0.19*). Also, similar to the heuristic that selects between 1-call-site-sensitive and 1-object-sensitive analyses (Figure 2d), the heuristic between 1-call-site-sensitive and 1st-parameter-sensitive analyses is dependent on the relationship between the context elements generated by both analyses. If 1st-parameter-sensitive analysis potentially generates many more context elements than 1-call-site sensitive analysis (i.e., *FC6 / FC2 > 3.8*), we expect the 1st-parameter-sensitive analysis to be more precise. Otherwise (i.e., *FC6 / FC2 <= 3.8*), depending on the importance of the first parameter to the function behavior, 1-call-site-sensitive analysis (when *0.19 < FC8 / FC3 <= 0.35*) or 1st-parameter-sensitive analysis (when *FC8 / FC3 > 0.35*) is selected. The balanced F-scores for 1-call-site-sensitive and 1st-parameter-sensitive analyses in this heuristic are 0.73 and 0.66, respectively.

1-object vs. 1st-parameter. Finally, Figure 2f presents the heuristic that selects between 1-object-sensitive and 1st-parameter-sensitive analyses. Function characteristics *FC4-FC8* are considered and the *prerequisite* is *FC5 > 0* and *FC7 > 0 OR FC8 > 0*. It is not surprising that 1-object-sensitive analysis is selected by the heuristic when the *this* object is more frequently used (i.e., *FC5 /FC8 > 1.34*) and 1st-parameter-sensitive analysis is selected when the condition is opposite (i.e., *FC5 /FC8 <= 0.8*). When uses of the *this* object and the first parameter are similar (i.e., *0.8 < FC5 /FC8 < 1.34*), the number of context elements generated by these two analyses decides the selection: (i) if 1-object-sensitive analysis potentially generates more context elements than 1st-parameter-sensitive analysis (i.e., *FC5 /FC 8 > 1*), we expect 1-object sensitive analysis to be more precise; (ii) if 1st-parameter-sensitive analysis generates more than twice the number of context elements than 1-object-sensitive analysis (i.e., *FC5 /FC 8 < 0.5*), 1st-parameter-sensitive analysis is selected; (iii) otherwise (i.e., *0.5 <= FC5 /FC 8 <= 1*), it is not clear from the data in the benchmarks which analysis produces more precise results because the function characteristics indicate that they have similar capability to analyze the function. In this case, we randomly select between 1-object-sensitive and 1st-parameter-sensitive analyses for the function whose characteristics fall in this region. The balanced F-scores for 1-object-sensitive and 1st-parameter-sensitive analyses in this heuristic are 0.79 and 0.86, respectively.

```
iParName < jParName: jth-parameter
iParName = jParName
|   iParOther < jParOther: jth-parameter
|   iParOther = jParOther
|   |   iParNum < jParNum: jth-parameter
|   |   iParNum >= jParNum: ith-parameter
|   iParOther > jParOther: ith-parameter
iParName > jParName: ith-parameter
```

Figure 3 Heuristic for ith-parameter vs. jth-parameter.

Summary. The heuristics presented in Figure 2 are intuitive for making a choice between each pair of analyses. More importantly, the heuristics for the call-strings approach and the functional approaches (i.e., Figures 2d and 2e) allow us to make a decision between two incomparable analyses. Finally, these heuristics are accurate (i.e., good balanced F-scores) in terms of their effectiveness on the benchmark programs.

4 Adaptive Context-sensitive Analysis

In this section, we present our adaptive context-sensitive analysis algorithm. This staged analysis (i) uses baseline analysis to obtain a call graph and points-to solution to extract function characteristics and (ii) performs an adaptive context-sensitive analysis based on heuristics that select an appropriate context-sensitive analysis for each function.

4.1 Function Characteristics Extraction

In Section 3, we discussed the function characteristics used in the heuristics to select from baseline, 1-call-site-sensitive, 1-object-sensitive and 1st-parameter-sensitive analyses. Various other context-sensitive analyses exist for improving analysis precision. For example, ith-parameter-sensitive analysis (Section 2.1) provides variations to distinguish function calls based on the computation states of parameters, decided by the parameter i.

In our adaptive context-sensitive analysis, we actually apply ith-parameter-sensitive analysis for a function whose precision relies on how accurately the ith parameter is analyzed. Therefore, for each parameter of a function, we extract three function characteristics: *iParNum*, *iParName* and *iParOther*. We apply the heuristics of 1st-parameter-sensitive analysis to select between ith-parameter-sensitive analysis and baseline(Figure 2c), 1-call-site-sensitive (Figure 2e) or 1-object-sensitive (Figure 2f) analysis. To select between ith-parameter-sensitive and jth-parameter-sensitive analyses, we apply the heuristic shown in Figure 3. We designed this heuristic based on the observation that for parameter sensitivity, a functional approach, the uses of the parameter whose computation states are used to distinguish function calls usually are more closely related to the analysis precision. In Figure 3, because the uses of a parameter as a property name in the dynamic property accesses is the most important characteristic, if one parameter is used as a property name more often than the other, distinguishing function calls based on its values may produce more precise results. If the *ParName* characteristics are the same for parameters i and j, the uses of the parameters in other situations are compared to decide if ith-parameter-sensitive analysis is more/less precise than jth-parameter-sensitive analysis. Finally, if both client-related metrics (i.e., *ParName* and *ParOther*) cannot distinguish the parameters, the heuristic selects the parameter that points to more objects.

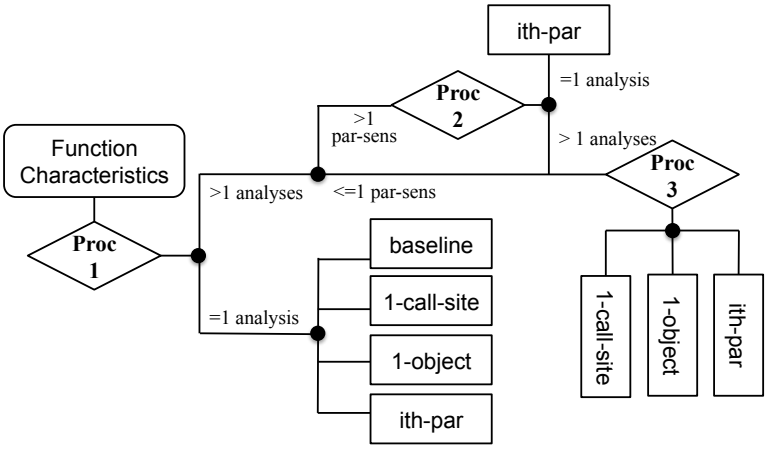


Figure 4 Workflow to select a context-sensitive analysis for a JavaScript function.

For a function *foo* with n parameters, we extract three characteristics for 1-call-site sensitivity (i.e., *CSNum*, *EquivCSNum* and *AllUse*), two characteristics for 1-object sensitivity (i.e., *RCNum* and *ThisUse*), and three characteristics for ith-parameter sensitivity (i.e., *iParNum*, *iParName* and *iParOther*). In total, there are *6+3n* function characteristics for *foo*.

4.2 Algorithm

Our adaptive context-sensitive analysis automatically selects a specific context-sensitive analysis for each function based on the function characteristics derived from the baseline analysis. Figures 2a-2f and 3 present the heuristics used to choose between pairs of analyses. The overall algorithm to select the context-sensitive analysis for a function is described in Figure 4. Given the function characteristics of function *foo*, Procedure 1 performs all pairwise comparisons between baseline analysis and 1-call-site-sensitive, 1-object-sensitive and ith-parameter-sensitive analyses for all the parameters of *foo*. If Procedure 1 returns a single analysis, this analysis is selected for *foo*. Returning baseline analysis means none of the context-sensitive analyses makes much difference to improve precision for *foo*.

In case more than one choice is returned by Procedure 1, further comparisons are conducted to decide the specific context-sensitive analysis to use for *foo*. If the analysis precision of *foo* may benefit from applying parameter-sensitive analyses on multiple parameter choices returned by Procedure 1, Procedure 2 selects from among them to find the parameter i that may produce the most precise results when ith-parameter-sensitive analysis is applied. If the choices from Procedure 1 are now narrowed down to only the ith-parameter-sensitive analysis, this analysis is selected by our algorithm to analyze *foo*.

When necessary, Procedure 3 chooses from the remaining context-sensitive analyses that are returned by Procedures 1 and 2. If there are two remaining context-sensitive analyses to choose from, Procedure 3 applies the heuristic in Figure 2d, 2e or 2f to decide on the context-sensitive analysis for analyzing *foo*. Otherwise (i.e., to choose from all three context-sensitive analyses), Procedure 3 compares each pair of 1-call-site-sensitive, 1-object-sensitive and ith-parameter-sensitive analyses and tries to find a best context-sensitive analysis for a majority of the pairs using heuristics in Figures 2d, 2e and 2f. For example, the adaptive analysis selects 1-call-site-sensitive analysis to analyze *foo* if it is chosen by both heuristics comparisons with 1-object-sensitive and ith-parameter-sensitive analyses. Finally, if Procedure 3 cannot

decide on a specific accurate context-sensitive analysis (i.e, when each of the three heuristics returns a different analysis choice), the adaptive analysis randomly chooses an analysis for *foo*.

5 Evaluation

In this section, we first present the details of our experimental setup. We then evaluate our adaptive context-sensitive analysis using two sets of benchmarks. We compared the precision of adaptive analysis to other context-sensitive analyses applied to the entire program.

5.1 Experiment Setup

Our implementation of adaptive context-sensitive analysis was based on the *WALA* static analysis infrastructure that supports JavaScript analysis. The baseline points-to analysis, *ZERO_ONE_CFA* analysis in *WALA* that uses the default context sensitivity for JavaScript analysis (Section 2.2), produced a call graph and a points-to solution from which we extracted the function characteristics. For the adaptive context-sensitive analysis, we implemented a new context selector[9] that applies the context-sensitive analysis chosen by the heuristics for each function. Note that the default context-sensitive analysis is always used as well to ensure that the results of adaptive analysis are comparable to the baseline analysis.

The goals of the experiments include: (i) comparing the precision of adaptive context-sensitive analysis with each of the other context-sensitive analyses to learn if the adaptive analysis improves JavaScript analysis precision and (ii) studying the accuracy of selecting a specific context-sensitive analysis for each function to validate the quality of the heuristics presented in Section 3.

To achieve these goals, we evaluated our analysis on two sets of benchmarks: (i) the same benchmark programs on which we performed the empirical study in Section 2.2 (i.e., Benchmarks I including the 28 JavaScript programs collected by Kashyap *et al.* [5], divided into four categories) and (ii) four open-source JavaScript applications or libraries (i.e., Benchmarks II). The programs in Benchmarks II are (i) *Box2DWeb*, collected in the *Octane* benchmarks[10], (ii) *minified.js* library[11] version 1.0, (iii) *mootools* library[12] version 1.5.1, and *benchmark.js* library[13] version 1.0.0. Because the heuristics were designed based on machine learning results using Benchmarks I, Benchmarks II serve to test if these heuristics can be applied by the adaptive context-sensitive analysis to arbitrary JavaScript programs and produce fairly accurate analysis results for the *Pts-Size* client.

5.2 Experimental Results

Results for Benchmarks I. Figure 5 shows the analysis precision results for Benchmarks I. We compared the results of our adaptive analysis with the context-sensitive analysis (i.e., 1-call-site-sensitive, 1-object-sensitive or 1st-parameter-sensitive analysis) that produced most accurate results for each program for these benchmarks. We define a context-sensitive analysis to be the *winner* analysis for a program if it was at least as precise as the other

[9] In *WALA*, the context element at a call site is decided by a context selector.
[10] `https://developers.google.com/octane/`
[11] `http://minifiedjs.com`
[12] `http://mootools.net`
[13] `http://benchmarkjs.com`

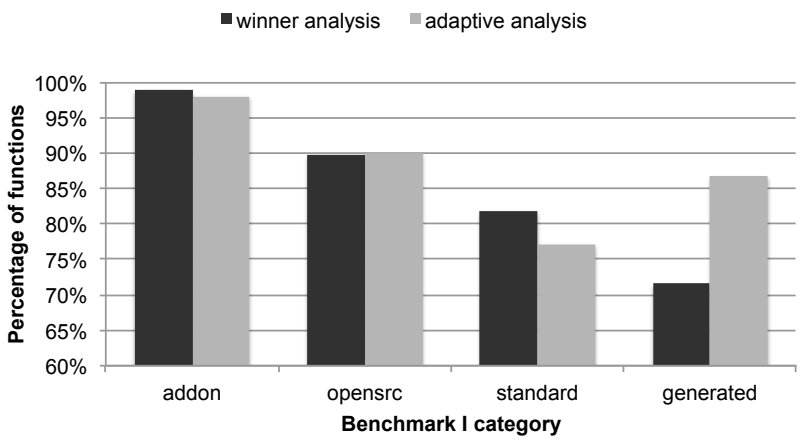


Figure 5 Analysis precision on Benchmarks I.

two context-sensitive analyses on the largest number of functions. For all 14 programs in the *addon* and *opensrc* benchmarks, 1-call-site-sensitive analysis was the *winner* among the 1-call-site-sensitive, 1-object-sensitive and 1st-parameter-sensitive analyses. For the *standard* benchmarks, 1st-parameter-sensitive analysis was *winner* on three programs and 1-call-site-sensitive analysis was *winner* on the other four programs. For all but one program in the *generated* benchmarks, 1-object-sensitive analysis was the *winner* analysis and 1-call-site-sensitive analysis was *winner* for *fourinarow*. In Figure 5, the *winner analysis* bar shows the percentage of the total number of functions in each benchmark category on which the *winner* analysis produced most accurate results. For example, the leftmost *winner analysis* bar represents that 1-call-site analysis (i.e., the *winner* analysis for all the programs in the *addon* benchmarks) produced at least as precise results as 1-object-sensitive and 1st-parameter-sensitive analyses for 98.8% of the functions in the *addon* benchmarks. The *adaptive analysis* bar shows the percentage of functions in each benchmark category for which our adaptive analysis produced at least as precise results as 1-call-site-sensitive, 1-object-sensitive and 1st-parameter-sensitive analyses.

In Figure 5, our adaptive analysis produced at least as precise results for most functions in the *addon* and *opensrc* benchmarks (i.e., 98% and 90%, respectively). In these two benchmark categories, 1-call-site-sensitive analysis was the *winner* analysis for all programs, producing at least as precise results for 98.8% and 89.8% of the functions, respectively. These results indicate our adaptive analysis is capable of producing similar precision for a set of programs that shares the same *winner* analysis. For the *standard* benchmarks, the *winner* analyses (i.e., 1-call-site-sensitive analysis for four programs and 1st-parameter-sensitive analysis for three programs) were more precise than the adaptive analysis in terms of the percentage of functions for which an analysis produced at least as precise results (i.e., 81.9% of the functions for the *winner* analyses comparing to 77.1% of the functions for adaptive analysis). Nevertheless, the adaptive analysis still achieved good precision for most of these relatively small programs in the *standard* benchmarks. Even though there were different *winner* context-sensitive analyses on the programs from the *standard* benchmarks, the use of the adaptive analysis avoided having to manually pick a specific context-sensitive analysis for an individual program. Finally, for the programs in the *generated* benchmarks, our adaptive analysis significantly improved precision over the *winner* context-sensitive analyses (i.e., 1-object-sensitive analysis for 6 programs and 1-call-site-sensitive analysis for the other one),

Table 2 Selection precision for Benchmarks I.

best / equally precise analysis	# of observed functions	# of selected functions (true positives)	true positive rate
1-call-site	351	258	73.5%
1-object	241	164	68.0%
1st-parameter	162	79	48.8%
1-call-site = 1-object	153	1-call-site: 39	94.1%
		1-object: 105	
1-call-site = 1st-parameter	39	1-call-site: 23	74.4%
		1st-parameter: 6	
1-object = 1st-parameter	6	1-object: 4	83.3%
		1st-parameter: 1	
1-call-site = 1-object = 1st-pararameter	25	1-call-site: 2	100%
		1-object: 13	
		1st-parameter: 10	
total	*977*	*704*	72.1%

from 71.7% to 86.7% functions. According to the results in Figure 1a, the programs in the *generated* benchmarks require context sensitivity for precision and moreover, these programs often benefited from different context-sensitive analyses. The results for the *generated* benchmarks show that we achieved our goal to analyze a multi-paradigm JavaScript program more accurately using a different context-sensitive analysis for each function.

The most important aspect of adaptive analysis is its ability to select an appropriate context-sensitive analysis for a specific function. Table 2 shows the accuracy of the analysis selection process for a function using the heuristics presented in Section 3 with the *Pts-Size* client. The first column in Table 2 lists the (set of) analyses that are *best* or *equally precise* (i.e., rows 4-7 in the first column) for a function (see Section 2.2). The second column shows the total number of functions in all programs from Benchmarks I on which the corresponding analyses were observed to produced the *best* or *equally precise* results. There were in total 1817 functions analyzed in Benchmarks I and the precision results of 977 functions were improved over baseline analysis by at least one context-sensitive analysis for the *Pts-Size* client. The last column presents the the number of functions on which the adaptive analysis matched the observed results (i.e., true positives for our heuristics). For those functions on which 1-call-site-sensitive and 1-object-sensitive analyses produced the *best* results, the selection heuristics resulted in good precision (i.e., 73.5% and 68%, respectively). However, the selection on 1st-parameter-sensitive analysis only achieved 48.8% precision. This is because our adaptive analysis chooses the appropriate ith-parameter-sensitive analysis to analyze a function using the parameter sensitivity. Here we are only checking the selection precision with respect to 1st-parameter-sensitive analysis; whereas ith-parameter-sensitive analysis (i>1) was applied to analyze 51 functions in the programs of Benchmarks I.

1-call-site-sensitive and 1-object-sensitive analyses produced *equally precise* results in terms on *Pts-Size* client on 153 functions. The adaptive analysis correctly selected 1-call-site-sensitive or 1-object-sensitive analysis to analyze 144 of those 153 functions, and interestingly, the choice was leaning towards 1-object-sensitive analysis (i.e., 1-object-sensitive analysis for 105 functions comparing to 1-call-site-sensitive analysis for 39 functions). For the functions on which *equally precise* results were produced by 1-call-site-sensitive and

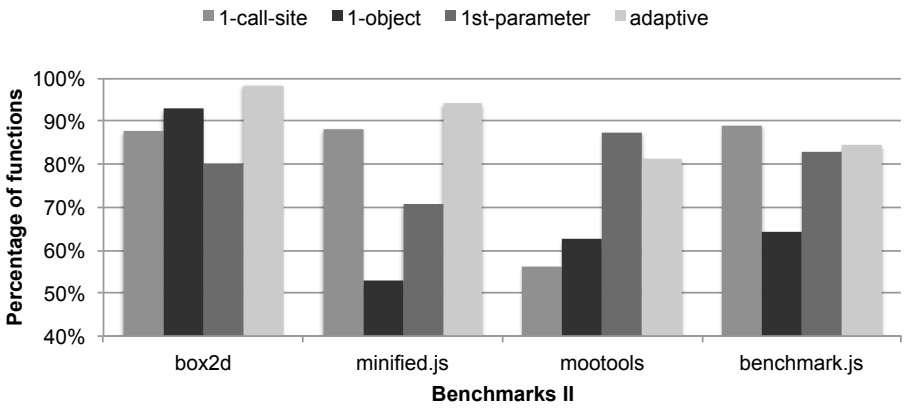


Figure 6 Analysis precision on Benchmarks II.

1st-parameter-sensitive analyses, adaptive analysis selects more functions to be analyzed by 1-call-site-sensitive analysis. The overall precision of selecting a context-sensitive analysis by our heuristics is very good (i.e., 72.1%); this is a measure of when adaptive analysis made the best choice possible. The above observations may help us to improve the heuristics in the future.

The time cost of our adaptive analysis is the sum of its two stages (i.e., the baseline points-to analysis to gather function characteristics and the subsequent adaptive context-sensitive analysis). We compare the performance of our adaptive analysis with the *winner* analysis for each program in Benchmarks I. On average over all the programs in Benchmarks I, our two-staged analysis introduced a 67% overhead. Nevertheless, the second stage (i.e., the adaptive context-sensitive analysis) is on average 19% faster than the *winner* analysis over the Benchmarks I programs. This result suggests that an appropriate choice of context sensitivity per function yields better performance and precision.

Results for Benchmarks II. Figure 6 shows initial analysis precision results using four programs from Benchmarks II. The sizes of these programs, in terms of the number of functions analyzed, are 126, 119, 80 and 64, respectively. The *1-call-site*, *1-object* and *1st-parameter* bars represent the percentage of functions on which each context-sensitive analysis produced at least as precise results as the other two. The *adaptive* bar (rightmost) represents the percentage of functions on which the adaptive analysis produced at least as precise results as 1-call-site-sensitive, 1-object-sensitive and 1st-parameter-sensitive analyses. We picked these four programs in Benchmarks II because their analysis results for different context-sensitive analyses were varied. For example, 1-object-sensitive analysis was more precise than 1-call-site-sensitive and 1st-parameter sensitive analyses for *box2d*, while 1st-parameter-sensitive analysis was the most precise for *mootools*.

The results in Figure 6 show that our adaptive analysis achieved better results than any single context-sensitive analysis for *box-2d* and *minified.js*. For example, 1-object-sensitive analysis was at least as precise for 92.8% of the functions in *box-2d*; adaptive analysis improved these results to 98.4% of the functions. 1-call-site-sensitive analysis produced at least precise results for 88.2% of the functions in *minified.js*; adaptive analysis improved the results by 5.9% more functions. 1st-parameter-sensitive and 1-call-site-sensitive analyses were the most precise context-sensitive analyses for *mootools* and *benchmark.js*, respectively. While adaptive analysis produced results lower than these analyses, the results of adaptive analysis

were close, only different for 6.2% and 4.7% of the functions in *mootools* and *benchmark.js*, respectively. Overall, our adaptive context-sensitive analysis was fairly accurate for analyzing these programs from Benchmarks II. This promising result indicates that the heuristics presented in Section 3 may be applicable in general to JavaScript programs.

5.3 Discussion

In this work, we have demonstrated the ability of our adaptive analysis that chooses a specific context-sensitive analysis for each function in order to significantly improve analysis precision. Nevertheless, this initial work has inspired us with more research ideas for further improvements of context-sensitive analyses for JavaScript. First, since we evaluated the adaptive analysis on a simple client of points-to analysis (i.e., *Pts-Size*), it would be interesting to know if adaptive analysis is effective to improve precision for other clients (e.g., security analysis). Second, context-sensitive analysis for JavaScript are not limited to 1-call-site, 1-object and ith-parameter. A deeper object-sensitive analysis (i.e., k-object) or another context-sensitive analysis (e.g., using the length of the parameter list at a call site as context element to distinguish JavaScript variadic functions [21]) could be used by adaptive analysis. New heuristics need to be designed for selecting these analyses. Third, we would like to explore if analysis precision may benefit from applying multiple-sensitive analyses on a specific JavaScript function. The idea of hybrid context-sensitive analysis has been tried for analyzing Java programs [6]. Fourth, scalability is an important issue for JavaScript analyses, especially for analyzing JavaScript websites that use libraries heavily (e.g., jQuery). In this work, we do not address this problem, that is, when a baseline points-to analysis is not scalable for a large JavaScript application, typically a website. In the future, we plan to focus on improving the performance of analysis of JavaScript websites using an adaptive approach.

Although we used benchmarks collected by Kashyap *et al.* [5] as well as other JavaScript programs to evaluate adaptive context-sensitive analysis, these programs may not be representative of non-website JavaScript applications, which might threaten the validity of our conclusions as applicable to all JavaScript programs.

6 Related Work

To the best of our knowledge, we have proposed the first analysis for JavaScript that adaptively uses multiple context-sensitive analyses to analyze a program when context sensitivity may help improve precision. Nevertheless, our work is related to approaches that apply context-sensitive analysis selectively (e.g., refinement-based analysis [3, 18, 17] and hybrid context-sensitive analysis [6]) for other programming languages. We already have discussed several context-sensitive analyses in Sections 1 and 2. In this section, we focus on related "selective" context-sensitive analyses.

Context-sensitive analysis has been deeply investigated for other object-oriented languages such as Java. However, these object-oriented languages do not seem as amenable to our approach of using different context-sensitive analyses on different functions. Castries and Smaragdakis presented hybrid context-sensitive points-to analysis for Java [6]. Several combinations of call-site and object-sensitive analyses were explored and evaluated for precision. Their results showed that selectively adding call-site-sensitive analysis to specific places in the program (e.g., static calls) significantly improved the precision of object-sensitive points-to analysis for Java. Our adaptive analysis automatically chooses an appropriate context-sensitive analysis for each function in JavaScript program.

Several works were proposed to tune the context sensitivity of an analysis based on pre-analysis results. Smaragdakis *et al.* presented introspective analysis that aims to improve the performance of a context-sensitive analysis for Java [17]. Introspective analysis selectively refines allocation sites or call sites based on the heuristics consisting of metrics computed from context-insensitive points-to results. The heuristics are tunable via constant parameters. In our adaptive analysis, the heuristics are computed from baseline analysis and syntactic analysis. The heuristics in our analysis focus on "which" context-sensitive analysis may improve precision instead of "if" context sensitivity would be of benefit.

Sridharan and Bodík presented a refinement-based points-to analysis for Java [18] that refines sensitivity for heap accesses and method calls. It also is demand-driven in that it skips irrelevant code in the analysis. Our adaptive context-sensitive analysis aims to improve precision for the whole program.

Guyer and Lin presented a client-driven analysis for C that automatically adjusts its precision in response to the needs of client analyses [3]. This client-driven analysis monitors polluting assignments (i.e., the program points that result inaccuracy in the analysis) and tunes context as well as flow sensitivity to improve precision. Liang and Naik presented another client-driven algorithm for Java that prunes away analysis results irrelevant to refinement for more precision [8]. For these techniques, a pre-analysis is used to determine the program points for refinement. Baseline points-to analysis is used to derive the function characteristics for our heuristics. Furthermore, our adaptive analysis involves more than one context-sensitive analysis.

Oh et al. presented a selective context-sensitive analysis for C guided by an impact pre-analysis [11]. The impact pre-analysis applies full context sensitivity (i.e., ∞-CFA) but with simplified abstract domain and transfer functions to infer the impacts of context sensitivity in the main analysis. The heuristics in our adaptive analysis focus on the characteristics of a function to indicate whether analysis precision for a function would benefit from a specific context-sensitive analysis. Our pre-analysis, the baseline analysis, is comparable with the adaptive analysis in terms of abstract domain and transfer functions.

7 Conclusions

The effectiveness of a context-sensitive analysis on a JavaScript program depends on its coding style because JavaScript features both object-oriented and functional programming paradigms. The fact that there is no winner context-sensitive analysis for the JavaScript benchmarks we examined motivated us to design an adaptive analysis. Our analysis applies a specialized context-sensitive analysis per function, using heuristics based on function characteristics derived from an inexpensive points-to analysis. Our experimental results show that adaptive analysis is more precise than any single context-sensitive analysis for several programs in the benchmarks, especially for those multi-paradigm programs whose analysis precision can benefit from multiple context-sensitive analyses. This work also has inspired opportunities to solve fundamental problems in analyzing JavaScript programs including analysis scalability for websites.

References

1 Ole Agesen. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, ECOOP'95, pages 2–26, 1995.

2 David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call graph construction in object-oriented languages. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA'97, pages 108–124, 1997.

3 Samuel Z. Guyer and Calvin Lin. Client-driven pointer analysis. In *Proceedings of the 10th International Conference on Static Analysis*, SAS'03, pages 214–236, 2003.

4 Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for JavaScript. In *Proceedings of the 16th International Symposium on Static Analysis*, SAS'09, pages 238–255, 2009.

5 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 *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 121–132, 2014.

6 George Kastrinis and Yannis Smaragdakis. Hybrid context-sensitivity for points-to analysis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'13, pages 423–434, 2013.

7 Ondřej Lhoták and Laurie Hendren. Evaluating the benefits of context-sensitive points-to analysis using a bdd-based implementation. *ACM Trans. Softw. Eng. Methodol.*, 18(1):3:1–3:53, October 2008.

8 Percy Liang and Mayur Naik. Scaling abstraction refinement via pruning. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'11, pages 590–601, 2011.

9 Henry Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *Conference proceedings on Object-oriented programming systems, languages and applications*, OOPLSA'86, pages 214–223, 1986.

10 Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Trans. Softw. Eng. Methodol.*, 14(1):1–41, January 2005.

11 Hakjoo Oh, Wonchan Lee, Kihong Heo, Hongseok Yang, and Kwangkeun Yi. Selective context-sensitivity guided by impact pre-analysis. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'14, pages 475–484, 2014.

12 J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

13 RedMonk. The RedMonk programming language rankings. `http://redmonk.com/sogrady/2014/06/13/language-rankings-6-14/`, 2014.

14 Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. *Program Flow Analysis: Theory and Applications*, pages 189–234, 1981.

15 Olin Grigsby Shivers. *Control-flow Analysis of Higher-order Languages of Taming Lambda*. PhD thesis, Carnegie Mellon University, 1991.

16 Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your contexts well: understanding object-sensitivity. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL'11, pages 17–30, 2011.

17 Yannis Smaragdakis, George Kastrinis, and George Balatsouras. Introspective analysis: Context-sensitivity, across the board. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'14, pages 485–495, 2014.

18 Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for Java. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'06, pages 387–400, 2006.

19 Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip. Correlation tracking for points-to analysis of JavaScript. In *Proceedings of the 26th European Conference on Object-Oriented Programming*, ECOOP'12, pages 435–458, 2012.

20 Peter Wegner. Dimensions of object-based language design. In *Conference proceedings on Object-oriented programming systems, languages and applications*, OOPSLA'87, pages 168–182, 1987.

21 Shiyi Wei and Barbara G. Ryder. Practical blended taint analysis for JavaScript. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 336–346, 2013.

22 Shiyi Wei and Barbara G. Ryder. State-sensitive points-to analysis for the dynamic behavior of JavaScript objects. In *Proceedings of the 28th European Conference on Object-oriented Programming*, ECOOP'14, pages 1–26, 2014.