Type Refinement for Static Analysis of JavaScript

Vineeth Kashyap[†] John Sarracino[†, ‡] John Wagner[†] Ben Wiedermann[‡] Ben Hardekopf[†]

[†]University of California, Santa Barbara [‡]Harvey Mudd College

vineeth@cs.ucsb.edu john_sarracino@hmc.edu john_wagner@umail.ucsb.edu benw@cs.hmc.edu benh@cs.ucsb.edu

Abstract

Static analysis of JavaScript has proven useful for a variety of purposes, including optimization, error checking, security auditing, program refactoring, and more. We propose a technique called *type refinement* that can improve the precision of such static analyses for JavaScript without any discernible performance impact. Refinement is a known technique that uses the conditions in branch guards to refine the analysis information propagated along each branch path. The key insight of this paper is to recognize that JavaScript semantics include many implicit conditional checks on types, and that performing type refinement on these implicit checks provides significant benefit for analysis precision.

To demonstrate the effectiveness of type refinement, we implement a static analysis tool for reporting potential type-errors in JavaScript programs. We provide an extensive empirical evaluation of type refinement using a benchmark suite containing a variety of JavaScript application domains, ranging from the standard performance benchmark suites (Sunspider and Octane), to open-source JavaScript applications, to machine-generated JavaScript via Emscripten. We show that type refinement can significantly improve analysis precision by up to 86% without affecting the performance of the analysis.

Categories and Subject Descriptors F.3.2 [*Semantics of Programming Languages*]: Program Analysis

1. Introduction

Dynamic languages have become ubiquitous. For example, JavaScript is used to implement a large amount of critical online infrastructure, including web applications, browser addons/extensions, and interpreters such as Adobe Flash. In response to the growing prominence of dynamic languages, the research community has begun to investigate how to apply *static analysis* techniques in this domain. Static analysis is used to deduce properties of a program's execution behavior; these properties can be used for a variety of useful purposes including optimization [21, 27], error checking [33], verification [11], security auditing [18, 19], and program refactoring [13], among other uses. However, dynamic languages present a unique challenge to static analysis, inherent in their very name: the dynamic nature of these languages makes creating precise, sound, and efficient static analyses a daunting task.

In this paper we focus on the static analysis of JavaScript, though in principle our proposed techniques are applicable to other dynamic languages as well. Our work is complementary to other recent work on JavaScript analysis, which has focused on understanding a program's types by proposing various novel abstract domains to track type information [24, 27]. This focus on types is essential for JavaScript analysis; because JavaScript behavior relies heavily on the runtime types of the values being operated on, understanding types is a necessary prerequisite to understanding many other properties of program behavior. However, with one exception (discussed further in Section 5) this prior work on JavaScript analysis has ignored an observation that has been profitably exploited in more traditional static analyses: that branch conditions (i.e., predicates that determine a program's control flow) necessarily constrain the set of values that can flow into the corresponding branches. This observation can be used to *refine* the abstract information propagated by the static analysis within each branch, thus improving the precision of the analysis. The details of how this concept works and how it can be applied to improve the precision of static analysis are explained in Appendix A (for any analysis in general) and Section 2 (for JavaScript analysis specifically).

While this general observation is well-known in the static analysis community, applying it specifically to JavaScript raises several important questions that must be answered to gain any useful benefit: (1) what kinds of conditions provide the most useful information for refinement; (2) how prevalent are these kinds of conditions in realistic JavaScript programs; and (3) how can we best exploit these conditions, based on their prevalence and usefulness, to substantially increase the precision of static analysis?

1.1 Key Insight

Our key insight that informs our proposed technique is that the most prevalent and useful conditional branches are not *explicit* in the text of JavaScript program, i.e., these conditions do not show up syntactically as **if** or **while** statements. Rather, they are *implicit* in the JavaScript semantics themselves. As an example, consider the statement `var result = myString.length;`. While syntactically there are no conditional branches in this statement, during execution there are several conditional branches taken by the JavaScript interpreter:

- Is `myString` either **null** or **undefined**? If so then raise a type-error exception, otherwise continue execution.
- Is `myString` a primitive value or an object? If it's a primitive value then convert it to an object first, then access the `length` property; otherwise just access the `length` property.
- Does the object (or one of its prototypes) contain a `length` property? If so then return the corresponding value, otherwise return **undefined**.

DLS '13, October 28, 2013, Indianapolis, Indiana, USA.

http://dx.doi.org/10.1145/2508168.2508175

Our thesis is that JavaScript static analysis can take advantage of these implicit conditional executions to refine the type information of the abstract values being propagated by the analysis, and that this *type refinement* can provide significant improvement in analysis precision.

1.2 Contributions

Our specific contributions are:

- A definition of type refinement for static analysis of JavaScript, including several variations that use different kinds of conditions to refine types (Section 2).
- An empirical evaluation of the proposed type refinement variations (Section 4). This evaluation is carried out on a more comprehensive set of JavaScript benchmarks than any presented in previous literature on JavaScript static analysis; it includes not only the standard SunSpider and V8 benchmark suites, but also a number of open-source JavaScript applications [1, 3] and a number of machine-generated JavaScript programs created using Emscripten [2].
- A set of recommendations for including type refinement in JavaScript analyses (Section 6). Our evaluation shows that taking advantage of implicit conditional branches provides a critical precision advantage for finding type errors, while the explicit `typeof` conditional branches exploited in previous work [20] provide only marginal benefit.

We conclude that type refinement is a promising technique for JavaScript analysis. This technique's design is informed by the semantics of JavaScript, enabling it to take advantage of language features hidden from plain sight and thus gain precision that would be lost by a technique that does not specifically exploit JavaScript semantics. Furthermore, type refinement is orthogonal to the question of designing abstract domains for JavaScript analysis; this means that it can profitably be combined with interesting new abstract domains in the future to achieve even better results.

2. The Potential for Refinement in JavaScript

Refinement allows an analysis to safely replace a less-precise answer with a more-precise answer. Appendix A gives suitable background on static analysis and the concept of refinement; readers unfamiliar with these notions may wish to refer to that appendix before continuing. Refinement can apply to many different abstract domains for analysis, but we hypothesize that, for JavaScript, the abstract domain of *types* is a particularly fruitful target for refinement. In JavaScript, as with many dynamic languages, the type of a value strongly influences the behavior of a program. Thus, refining type information intuitively would seem likely to improve the precision of JavaScript static analysis (and our empirical results bear out this intuition).

This observation means that we should focus our attention on those conditionals in the JavaScript program that are based on type information, i.e., conditionals whose truth or falsity constrain the set of types allowed in the corresponding branches. An obvious candidate is the set of conditionals that use the **typeof** operator to test value's types. For example, consider the following code:

```
if ( typeof x == "number" ) { x = x + 42; }
```

Suppose that immediately before the conditional, the static analysis has computed that `x` may be a number or a string. Then inside the true branch of the conditional, we can safely refine the type of `x` to be a number. This strategy is similar to the one employed by Guha et al [20] (discussed further in Section 5), though they were attempting to typecheck a subset of JavaScript rather than to improve the precision of JavaScript static analysis.

2.1 Key Insight

While the **typeof** check is an obvious candidate for refinement, our key insight is that most of the conditionals involving types aren't even syntactically present in the JavaScript program—rather, they are implicit in the semantics of the JavaScript language itself.

Consider the following statement:

```
var x = myString[i];
```

This seemingly simple statement requires a large number of implicit type checks. Example 1 makes all of these checks explicit. None of these checks involve **typeof**. Instead, we see three new kinds of conditions that involve type information.

Example 1 The semantics of `var x = myString[i];`

```
 1: if myString is null or undefined then
 2:     type-error

 3: else
 4:     // convert myString to an object first?
 5:     if myString is a primitive then
 6:         obj = toObject(myString)
 7:     else
 8:         obj = myString
 9:     end if

10:     // convert i to a string

11:     // case 1: i is a primitive
12:     if i is a primitive then
13:         prop = toString(i)
14:     else
15:         if i.toString is callable then
16:             tmp = i.toString()
17:         else
18:             goto line 26
19:         end if
20:     end if

21:     // case 2: i is not a primitive, but i.toString() is
22:     if tmp is a primitive then
23:         prop = toString(tmp)

24:     // case 3: i.toString() is not a primitive; try i.valueOf()
25:     else
26:         if i.valueOf is callable then
27:             tmp2 = i.valueOf()
28:         else
29:             type-error
30:         end if

31:         if tmp2 is a primitive then
32:             prop = toString(tmp2)
33:         else
34:             type-error
35:         end if
36:     end if

37:     // retrieve the property from the object
38:     x = obj.prop
39: end if
```

One condition (e.g., at line `1` in Example 1) checks whether a value is either `null` or `undefined`. JavaScript performs this check whenever a program attempts to access a property of a value; if the value is `null` or `undefined` it is a type-error. JavaScript also performs this check whenever a program attempts to add, update,

or delete a property of some value. We abbreviate this condition as **isUndefNull**.

Another condition (e.g., at lines 5, 12, 22, and 31) checks whether a value is primitive rather than an object, i.e., that it is either a number, a boolean, a string, `undefined`, or `null`[1]. JavaScript performs this check whenever the runtime might need to implicitly convert a value into another type. We abbreviate this condition as **isPrim**.

A third condition (e.g., at lines 15 and 26) checks whether a value is callable (i.e., that it is actually a function). If so, then the runtime calls the function; otherwise it can throw a type error exception. We abbreviate this check for callable as the **isFunc** condition.

The key insight of this paper is to focus refinement on those *implicit conditionals*—**isPrim**, **isUndefNull**, and **isFunc**—which abound in JavaScript programs.

2.2 Refinement on Implicit Conditions

JavaScript's implicit conditions restrict the types of values that flow along their branches. Refinement can take advantage of these restrictions as follows:

- **isPrim**: On the true branch, the checked value *must* be a primitive value; on the false branch it *must* be an object.
- **isUndefNull**: On the true branch, the checked value *must* be either `undefined` or `null`; on the false branch it *cannot* be `undefined` or `null`.
- **isFunc**: On the true branch, the checked value *must* be a function; on the false branch it *cannot* be a function.

The benefits become evident when we consider a static analysis that does *not* use refinement for these conditions. For **isPrim** the benefit comes from the false branch of the conditional, for example, line 15 in Example 1. Suppose that on line 12 the analysis computes that `i` may be either `undefined` or an object. In the false branch, `i`'s properties are accessed to make method calls (e.g., the `.toString` and/or `.valueOf` methods used to convert objects to primitives). However, since `i` may be `undefined`, the analysis conservatively computes that these calls may raise a type error exception. If `i` had been refined, then the analysis would know that it cannot be `undefined` on that branch, and hence there cannot be a type error exception.

The benefit for **isUndefNull** and **isFunc** is more subtle. Consider the following program fragment:

```
delete obj.p1;
obj.p2 = 2;
```

Example 2 The semantics of `delete obj.p1; obj.p2 = 2;`

```
 1: if obj is null or undefined then
 2:     type-error
 3: else
 4:     delete obj.p1
 5:     if obj is null or undefined then
 6:         type-error
 7:     else
 8:         obj.p2 = 2
 9:     end if
10: end if
```

The implicit behavior of this fragment is described by the pseudocode in Example 2.This example illustrates how implicit checks and exceptions can lead to spurious type-errors. Concretely, the code performs two sequential property modifications. If `obj` is `null` or `undefined`, the first statement causes a type-error, and the second statement never executes. Otherwise, both statements execute successfully. An analysis that uses refinement can capture this behavior, while an analysis that does not use refinement cannot, as explained below.

Consider an analysis of this fragment that has imprecise information: `obj` might be `null` or an object. In this case, the **isUndefNull** conditions in lines 1 and 5 of Example 2 are non-deterministic, and the analysis must conservatively propgate `obj`'s type to both branches. An analysis that does not use refinement must then conservatively report that two type-errors might occur: at lines 2 and 6. In reality, if the first statement of the program fragment successfully executes, so will the second. Refinement can detect this invariant: in the false branch of lines 4–9, the type of `obj` cannot be `null` or `undefined`. The **isUndefNull** condition at line 5 is therefore deterministic, so the analysis will not follow the branch to the error at line 6. Thus, a refined analysis can give the most precise result for imprecise data: if a type-error occurs, it occurs only as a result of the `delete` statement.

The **isFunc** check is similar to the **isUndefNull** check in that the cost comes from potentially passing unrefined values to successor nodes, causing the analysis to conservatively compute type error exceptions whereas an analysis using refinement would not. In general, the benefit of refinement in the presence of implicit exceptions is potentially tremendous: When exploring the path along which the exception does not occur, the analysis can refine the type of the value so that it does not cause any more implicit exceptions along that path. Our empirical evaluation demonstrates that, if an analysis focuses on these simple implicit type checks, refinement can dramatically increase the precision of a type-error analysis.

$$\sigma \in Store : Variable \to \mathcal{P}(Type)$$
$$PropertyMap : Property \to \mathcal{P}(Type)$$

$$\tau \in Type : PrimType + ObjType + FuncType$$
$$PrimType : \textbf{num} + \textbf{bool} + \textbf{str} + \textbf{null} + \textbf{undefined}$$
$$ObjType : PropertyMap$$
$$FuncType : Closure \times PropertyMap$$

Figure 1: A simplified version of an abstract domain suitable for type refinement. The abstract domain $Store$ maps variables to their abstract types. The abstract domain $ObjType$ maps object properties to a set of possible types. The abstract domain $FuncType$ includes a closure (the function to be called) and a property map (to model the function object).

3. Refining Types in JavaScript Analyses

The previous section discusses type refinement at a conceptual level. In this section we make the discussion concrete, describing specifically how we perform type refinement for JavaScript. Type refinement takes place in the context of some particular static analysis, however type refinement itself is largely independent of that surrounding context. Therefore we describe type refinement using a generic type-based abstract domain that would be common to any JavaScript static analysis, which in the actual analysis can be augmented to provide whatever additional information is relevant.

[1] Confusingly, **typeof** `null` == "object", but `null` is not an object.

$$x \in \mathit{Variable} \qquad p \in \mathit{Property}$$

$$\begin{aligned} a \in \mathit{Access} &::= x \mid x.p \\ c \in \mathit{Condition} &::= \textsf{typeof}(a) = \mathit{tag} \mid \textsf{isFunc}(a) \\ &\quad \mid \textsf{isUndefNull}(a) \mid \textsf{isPrim}(a) \\ \mathit{tag} &::= \textsf{"number"} \mid \textsf{"boolean"} \mid \textsf{"string"} \\ &\quad \mid \textsf{"undefined"} \mid \textsf{"object"} \mid \textsf{"function"} \end{aligned}$$

Figure 2: Type-based conditions for refinement. These conditions precisely describe the conditonal expressions that trigger refinement. An *access* is a low-level primitive—the simplest form of a variable or property access. Our analysis can handle any conditional expression that reduces to this form.

$c \in \mathit{Condition}$	$\mathit{filter}(c)$
$\textsf{typeof}(a) = \textsf{"number"}$	$\{\textsf{num}\}$
$\textsf{typeof}(a) = \textsf{"boolean"}$	$\{\textsf{bool}\}$
$\textsf{typeof}(a) = \textsf{"string"}$	$\{\textsf{str}\}$
$\textsf{typeof}(a) = \textsf{"undefined"}$	$\{\textsf{undefined}\}$
$\textsf{typeof}(a) = \textsf{"object"}$	$\{\textsf{null}\} \cup \mathit{ObjType}$
$\textsf{typeof}(a) = \textsf{"function"}$	$\mathit{FuncType}$
$\textsf{isFunc}(a)$	$\mathit{FuncType}$
$\textsf{isUndefNull}(a)$	$\{\textsf{undefined}, \textsf{null}\}$
$\textsf{isPrim}(a)$	$\{\tau_p \in \mathit{PrimType}\}$

Figure 3: Filters for refinement conditions. The analysis uses these filters to refine information along a condition's branches.

3.1 Type-based Abstract Domain

We now describe the abstract domain that we will be using to describe type refinement. A JavaScript value can be a primitive value (i.e., `number`, `boolean`, `string`, `undefined`, or `null`), an object, or a function[2]. The abstract domain of Figure 1 describes an approximation of these types. This abstract domain is deliberately simpler than one that would be used in an actual analysis, in order to make the exposition more clear by focusing on the aspects relevant to type refinement. A specific static analysis would augment this abstract domain with more information relevant to the purpose of that analysis (for examples of such augmented abstract domains, see [24, 27]).

The abstract domain in the figure represents the relevant type information that is propagated by the analysis from program point to program point. An abstract store Store maps variable names to sets of abstract types. We uses sets of types because, as discussed in Section A, the analysis is approximating the concrete program behavior—e.g., the analysis may be able to determine that a variable is either **num** or **undefined**, but not be able to narrow the type information down any further. An abstract object type $\mathit{ObjType}$ maps property names to their abstract types. An abstract function type $\mathit{FuncType}$ consists of a *closure* (the function to be called) and a property map (to model the fact that JavaScript functions are also objects).

A type-based static analysis operates over this abstract domain. Abstract stores flow along the program's control-flow graph (where each node is a program statement), and at each statement the analysis interprets the effect of that statement relative to a specific input σ flowing from that statement's predecessors, in order to determine the new σ' that is the output of that statement, which is then passed to that statement's successors. If the analysis encounters a type-based condition, the analysis may be able to increase the precision of the information contained in the store by refining the type information based on the condition, as described below.

3.2 Identifying Relevant Type-based Conditions

When using type refinement, the analysis interprets a branch condition as a *filter* along each branch of the conditional; these filters are used to refine the type information of the stores passed to each respective branch. In theory, *any* branch condition that constrains the types contained in the store can be used to perform type refinement. However, in practice some conditions are much more complicated to translate into filters than others. Therefore the analysis designer must make a tradeoff, by syntactically restricting the set of conditions from which the analysis extracts filters. The goal is to balance the additional precision that may be gained by interpreting certain conditions against the complexity of generating filters from those conditions.

Figure 2 shows the tradeoff that we have made in our type refinement implementation. The figure gives a restricted syntax for branch conditions (which also makes certain implicit checks explicit in the syntax rather than implicit in the language semantics). Our analysis only attempts refinement using conditions that are contained in this restricted syntax; any other conditions are treated the same as if the analysis were not doing type refinement. We chose this syntax to match the categories of explicit and implicit type checks from Section 2. The **typeof** condition corresponds to implicit and explicit checks on type equality. The **isFunc** condition corresponds to the implicit check for whether a value is a function. The **isUndefNull** condition corresponds to the implicit check that the JavaScript interpreter performs when accessing or modifying an object property. The **isPrim** condition corresponds to the implicit check that the JavaScript interpreter performs as part of implicit type conversion.

Each condition contains exactly one *access*. An access can have two forms: a variable or a *direct* property access. A direct property access $x.p$ gives the precise property name being accessed. The program need not literally contain a direct access; the program might specify the property access using a complex expression. As long as the static analysis can recover the direct access from the expression, the analysis will attempt to apply refinement. In practice, for **typeof** conditions, our analysis handles any conditions of the form **typeof** e_1 **==** e_2, where e_1 and e_2 are arbitrary JavaScript expressions. Our analysis currently does not handle more complex expressions than these. In particular, it does not handle logical combinations of these conditions. In Section 4, we demonstrate that the conditions in Figure 2 are sufficient to achieve significant increases in the precision of a type-based analysis; creating useful filters for more complicated expressions is left to future work.

3.3 Filtering Type Information

Each condition induces a *filter* that captures the types described by that condition. Figure 3 defines the filter for each possible condition. These filters match the description of the explicit and implicit information encoded in the scenarios from Section 2. The analysis uses a condition's filters to refine the values that flow along the condition's branches. Specifically, for each type-based condition c:

1. The analysis interprets condition c relative to an input store σ, to determine which branches (i.e, the true and false branches) to execute.

[2] Functions are also objects, but we distinguish them separately because some implicit checks are specific to functions.

2. The analysis uses the input store σ to retrieve the abstract type τ of the condition's access a.
3. When the analysis executes the true branch, it computes a new abstract type for a as follows: $\tau \cap \mathit{filter}(c)$. In other words, it intersects the current set of possible types for a with the set of possible types for a allowed by the branch condition. The analysis updates the type for a in σ and sends the updated store along the true branch.
4. When the analysis executes the false branch, it computes a new abstract type for a as follows: $\tau - \mathit{filter}(c)$. In other words, it removes from the current set of possible types for a those types which would have meant that the branch condition was true. The analysis updates the type for a in σ and sends the updated store along the false branch.

For example, suppose the analysis reaches a condition $\textsf{isUndefNull}(a)$ with a store that maps a to the abstract type $\{\textsf{undefined}, \textsf{num}\}$. In this case the condition evaluates to both **true** and **false**, and so the analysis must execute both the true and false branches. Along the true branch, the analysis sends a store that assigns a the type $\{\textsf{undefined}, \textsf{num}\} \cap \mathit{filter}(\textsf{isUndefNull}(a)) = \{\textsf{undefined}\}$. Along the false branch, the analysis sends a store that assigns a the type $\{\textsf{undefined}, \textsf{num}\} - \mathit{filter}(\textsf{isUndefNull}(a)) = \{\textsf{num}\}$.

3.4 Sound Type Refinement

Type refinement is *sound* if and only if the filtered set of types sent to a branch is a superset of all types that might ever be seen at that branch over all possible concrete executions. Our refinement rules are sound as long as they are only applied to accesses that correspond to a single concrete access. This is the standard static analysis issue of *strong* vs *weak* updates: a strong update can replace a value with a completely new value (potentially more precise than the previous value), while a weak update can only replace a value with a weaker (i.e., less precise) value. This issue is best explained by example.

Suppose that a variable x is mapped by the abstract store to the set of types $\{[\text{foo} \mapsto \{\textsf{num}, \textsf{str}\}], [\text{foo} \mapsto \{\textsf{num}, \textsf{bool}\}]\}$. This abstract value means that the type of x may be one or the other of the two object types, but the analysis does not know which one. Now suppose that the analysis is considering a branch condition $\textsf{typeof}(x.\text{foo}) = \textsf{"number"}$. In the true branch, the type of x.foo must be **num**. However, the analysis does not know *which* of the two possible object types for x is correct, so it cannot determine which of the two object types has been constrained by the condition. Thus, the analysis cannot refine either object type because if it refines the wrong one, the analysis becomes unsound. When this is the case, any update to the abstract value for x must be weak: an analysis cannot replace the abstract value of x with a more precise version.

If, on the other hand, x refers to only a single possible object type, e.g., $\{[\text{foo} \mapsto \{\textsf{num}, \textsf{str}\}]\}$, then an analysis knows that this is the type constrained by the condition. Thus, the analysis can safely refine x's value in the true branch to $\{[\text{foo} \mapsto \{\textsf{num}\}]\}$. When this is the case, an update to the abstract value for x can be strong: the analysis *can* replace x's abstract value with a more precise value. Our analysis applies type refinement—which overwrites abstract values—only when the refinement corresponds to a strong update.

4. Evaluation

We have implemented our proposed ideas and evaluated their effect on a static analysis for JavaScript that detects potential type-error exceptions. We find that an analysis that performs type refinement on all of the conditions described in Section 3.2 can achieve a significant increase in analysis precision, with a minimal impact on the analysis performance. In this section we demonstrate the effectiveness of our ideas by comparing the type-error analysis *with* type refinement relative to the same analysis *without* type refinement, for a variety of JavaScript programs.

4.1 JavaScript Analysis Framework

We use the JSAI JavaScript static analyzer for our experiments. The source code for JSAI can be found at `http://www.cs.ucsb.edu/~pllab` under Downloads. JSAI is implemented in Scala version 2.10.1 using the Rhino parser as a front-end [5]. JSAI does not currently handle `eval` and related mechanisms for dynamic code injection, however none of our benchmarks use these mechanisms.

4.2 Benchmark Suite

JavaScript can be written in a number of different styles, and these styles can affect the usefulness of our type refinement technique. In order to explore this issue, we select benchmarks from a variety of application domains. We group the selected bencharks into three categories:

- **standard**: These are the standard benchmark suites, SunSpider [6] and Octane [4], that are used by browser vendors to test the correctness and performance of their JavaScript implementations.
- **opensrc**: These are real-world, handwritten JavaScript programs taken from various open source projects such as LINQ for JavaScript [3] and Defensive JS [1].
- **emscripten**: These are machine-generated JavaScript code, obtained by compiling C/C++ programs using the Emscripten [2] LLVM→JavaScript compiler.

We select seven benchmarks from each category for our evaluation. This benchmark suite is available for download[3]. The benchmarks exercise a wide range of JavaScript features, including core objects and APIs, typed arrays, etc. However, none of these benchmarks contain `eval` or equivalent features that allow dynamic code injection.

Figure 4 shows the distribution of program sizes in each benchmark category, based on the number of AST nodes created for the programs by the Rhino parser. We use AST nodes as the metric for program size because it correlates with the amount of work done by the analysis, which operates over AST nodes. The standard benchmarks, while not large, exercise several key features of the language, and we use them to test the correctness of our implementation. The opensrc and emscripten benchmarks are significantly larger, however it should be noted that the emscripten benchmarks contain a large amount of unreachable code because the Emscripten compiler automatically includes a large amount of unused library code.

4.3 Experimental Methodology

Our base analysis is flow-sensitive and context-sensitive, using a stack-based 1-CFA context-sensitivity strategy (i.e., it distinguishes function contexts by the callsite from which the function was invoked). The heap model uses static allocation sites to model abstract addresses. Starting from this base type-error analysis, we implement new analyses that incrementally add support for refining various kinds of conditions. We implement and evaluate a total of four type-error analyses:

- **B**: a base flow-sensitive, context-sensitive type-error analysis that does not perform refinement.

[3] Available with the rest of the repository under Downloads at `http://www.cs.ucsb.edu/~pllab`.

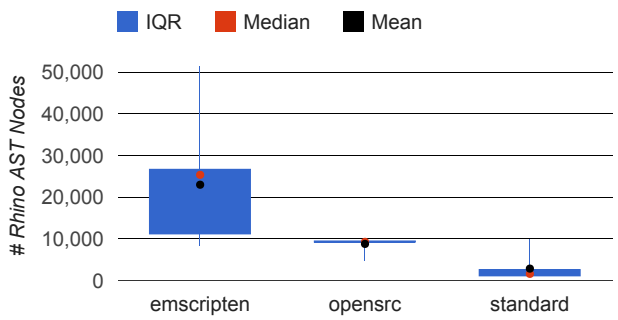


Figure 4: Graph to show size distribution (along y-axis) of benchmarks in each category (x-axis). Size is measured in terms of number of JavaScript AST nodes created by the Rhino parser [5]. For each benchmark category, the blue box gives the 25%-75% quartiles, the blue line gives the range of sizes, and median and mean are denoted by red and black dots respectively.

- **T**: the **B** analysis, extended with type refinement for conditionals that contain **typeof** checks.
- **TP**: the **T** analysis, extended to include type refinement for conditionals that contain **isPrim** checks.
- **TPUNF**: the **TP** analysis extended to include type refinement for conditionals that contain **isUndefNull** and **isFunc** checks.

We compare the precision and performance of these analyses for all the benchmarks in our suite. The metric we use to measure precision is the number of program locations (i.e., AST nodes) that the analysis computes may potentially throw type-error exceptions. The analysis that reports the fewest locations is the most precise. This metric correlates with the usefulness of a static type-error reporting tool: although false positives are inherent in a static analysis, the fewer the number of reported errors (i.e., the fewer the false positives) the more useful the tool is.

The metric we use to measure performance is execution time in seconds. We perform a trial for each (analysis, benchmark) pair. Each trial runs in its own invocation of the JVM. A trial starts with a warm-up run whose results are discarded. We then perform 10 runs in sequence and report the mean execution time of these 10 runs. All our experiments execute on an Ubuntu 12.04.2 LTS machine with CPU speed of 1.9GHz and 32GB RAM on JVM version 1.7.

4.4 Potential Opportunity for Type Refinement

In this section, we explore the potential benefits of type refinement across our benchmark categories. Type refinement is potentially useful for a given branch condition when the analysis treats that condition as non-deterministic—i.e., the analysis cannot determine for certain which branch is taken, and so must execute both branches. To gain an understanding of how many opportunities various flavors of type refinement can take advantage of in these benchmarks, we distinguish three kinds of branches:

- **T**: Branches with a **typeof** check in them.
- **P**: Branches with a **isPrim** check in them.
- **UNF**: Branches with a **isUndefNull** or **isFunc** check in them.

We also qualify each kind of branch to be:

- **D**: a deterministic branch.

category	branch kind	D	NDC	NDNC
standard	**T**	469	87	104
	P	2408	141	0
	UNF	5692	571	0
opensrc	**T**	408	82	50
	P	2048	80	0
	UNF	9456	374	0
emscripten	**T**	149	14	9
	P	595	3	0
	UNF	7120	12	0

Table 1: The table that shows for each category of benchmarks, the kind of branches that the analysis encounters. The numbers represent number of program locations. The abbreviations are further detailed in Section 4.4. The way to interpret this table is as follows: for example, the number under column **NDC**, and row **T** represents the number of program locations with branches that have **typeof** checks in them, and are non-deterministic and match our grammar for type refinement.

- **NDC**: a non-deterministic branch where the branch condition follows our restricted syntax for type refinement, and therefore is a candidate for our type refinement.
- **NDNC**: a non-deterministic branch where the branch condition does not match our restricted syntax for type refinement, and therefore is not a candidate for our type refinement.

Deterministic branches **D** provide no opportunity for type refinement at all. Non-deterministic, non-candidate branches **NDNC** could potentially benefit from type refinement if we extended our technique to include more complicated branches, but do not benefit from our current type refinement implementation. The non-deterministic candidate branches **NDC** are the branches that can benefit from our current implementation of type refinement.

We provide the above information about each qualified kind of branch for each benchmark category, using the **B** version of the analysis, and summarize the data in Table 1. The data shows that the deterministic branches far exceed the non-deterministic ones. This might seem surprising, but the reason is because the analysis must perform a number of checks that are almost always trivially true. For example, in the code `a[0] = 0`, the analysis checks that `a` is not `undefined` or `null`, and in a vast majority of cases this is true, making this conditional deterministic. This is particularly true of the emscripten benchmarks, which makes sense because they were generated from statically-typed languages. Although most branch points are deterministic, there are still a significant number of non-deterministic branches that can be exploited by type refinement.

4.5 Effects of Various Type Refinements

Table 2 presents the raw data of our evaluation, including the total number of type errors reported for each benchmark and the mean runtimes for the **B** and **TPUNF** analyses. Figure 5 extracts and summarizes the precision results from Table 2.

For the standard and opensrc benchmarks, the **T** configuration yields almost no benefit over the **B** configuration, meaning that doing type refinement over this kind of condition is not useful for reducing type-error exceptions. However, the **TP** configuration does yield a fair amount of benefit, and the **TPUNF** configuration yields significant benefit. For example, on the `cryptobench` benchmark in the standard category the **TPUNF** configuration reports 253 fewer type errors than the **B**, and for `rsa` in opensrc category, 124 fewer errors are reported by **TPUNF** configuration when compared to **B**.

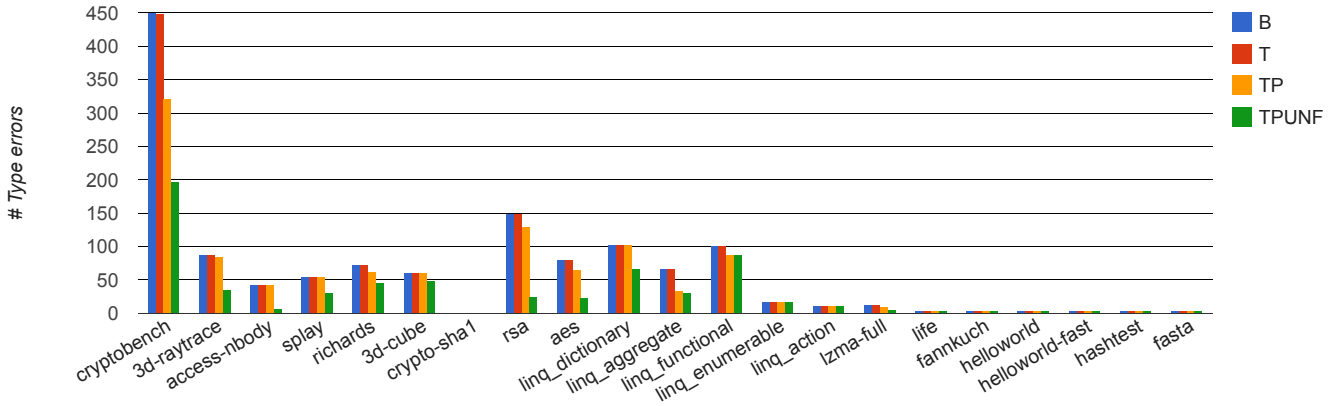


Figure 5: Analysis precision (in number of reported type-errors) with and without refinement; lower is better. Benchmarks are grouped by category. Under the **TPUNF** analysis with refinement, many of the standard benchmarks (`cryptobench` to `crypto-sha1`) and the opensrc benchmarks (`rsa` to `linq_action`) achieve more than 50% improvement in precision, relative to the **B** analysis without refinement.

In general, across benchmarks in the standard and opensrc categories, most of the benefits arise from type refinement for **isPrim**, **isUndefNull** and **isFunc** implicit checks.

For the emscripten benchmarks, type refinement does not make a significant impact on precision—this is expected based on the low potential available in these benchmarks for type refinement (see Figure 1), and because there are very few type-errors reported in the **B** version already. Because emscripten benchmarks are generated by compiling from a statically-typed language, the values in the program tend to be very monomorphic.

From Table 2, specifically the column giving the percentage increase in time, we observe that type refinements have a negligible effect on performance.

5. Related Work

Static analysis of dynamic languages is an active research area. Recent innovations include: a type analysis that relies on an abstract domain that is highly tuned for JavaScript [24]; various static type inference algorithms [9, 15] and hybrid type inference algorithms [8], including those that prevent access to undefined fields [33], enable program optimizations [21, 27], or are suitable for IDEs [28]; static analyses to secure the Web [17–19]; alias analyses for JavaScript [23, 30] and Python [16]; an analysis to support JavaScript refactoring [13]; and analysis frameworks for dynamic languages [7, 26]. All these techniques are orthogonal to type refinement. As such, they all may benefit from the idea of refinement in general, and the analyses for JavaScript would benefit directly from our contribution.

Two existing techniques for type inference of dynamic languages rely in particular on a notion of refinement: flow typing [20] and occurrence typing [31, 32]. Flow typing is a technique for JavaScript type inference that uses type tags in explicit `typeof` conditionals to filter type information [20]. Occurrence typing is another technique for refining types in branches based on the conditionals that govern those branches [31, 32]. Occurrence typing takes into consideration a more complex set of filters than flow typing and type refinement, including the effects of selectors (e.g., `car`). These filters are encoded in a propositional logic that forms the basis of a type system for Scheme.

Our work, type refinement, leverages a similar insight as does flow typing and occurrence typing, namely that runtime types in a dynamically typed language are constrained by branch conditions. Perhaps the most important distinction between our work and prior work is the focus on *which* branch conditions are mined by the analysis to obtain more precise information. Flow and occurrence typing focuses on branches that occur in common coding idioms: explicit behavior, encoded by a programmer, that follows a particular pattern. The reasoning behind this focus is that the programmers are communicating information by using an idiom, and automated understanding can take advantage of this high-level semantic information. Type refinement focuses on implicit behavior driven by the runtime: behavior encoded not by the programmer, but by the semantics of the language itself. The reasoning behind this focus is that implicit behavior (i.e., the semantics of the language) appears in *every* program, and if the analysis can take advantage of this behavior the potential benefit can be huge. In the context of type errors, our study shows that a focus on the implicit JavaScript behavior provides more benefit that the focus on only explicit JavaScript behavior.

Type refinement also differs from prior work in its goals and methods. The goal of flow and occurrence typing is a sound static type system for a dynamic language. Hence, Guha et al and Tobin-Hochstadt et al focus on type soundness (although Guha et al also validate their work against a corpus of Scheme code). In contrast, the goal for our work is to provide more precise information for JavaScript static analyses, which we validate by performing an evaluation of several variants against a large corpus of JavaScript programs. Operationally, our work differs from prior work in that prior work encodes the extra type information in the abstract domain, explicitly representing the information and essentially delaying the refinement. Ours work filters the results along a path, essentially applying the refinement immediately.

Type refinement should not be confused with *refinement types* [11, 14]. Although they have similar names and share other superficial similarities, they are different concepts. Type refinement is an action that occurs during program analysis: it filters the analysis values that flow along paths in the program's abstract execution. An analysis with type refinement is more precise than one without it. A refinement type is an entity, placed in a program by its author to express a restriction on the set of values that can be computed by a particular expression. A type system with refinement types can prove stronger properties about programs than one without them.

6. Conclusion and Future Work

We show in this work that type refinement is a useful precision optimization for static analysis of JavaScript. In particular, type refine-

Benchmark	Number of TypeErrors				Reduction in	Mean Runtime (s)		Slowdown
	B	**T**	**TP**	**TPUNF**	**TypeErrors**	**B**	**TPUNF**	
cryptobench	450	448	321	197	56%	47.46	46.30	−2.4%
3d–raytrace	87	87	84	35	60%	3.92	4.07	4%
access–nbody	43	43	43	6	86%	0.41	0.41	0%
splay	55	55	55	31	44%	0.78	0.78	0%
richards	73	73	62	45	38%	2.23	2.44	9%
3d–cube	60	60	60	49	18%	3.77	3.72	−1%
crypto–sha1	0	0	0	0	0%	0.30	0.29	−3%
standard	**768**	**766**	**625**	**363**	**53%**	**8.41**	**8.29**	**−1.4%**
rsa	148	148	129	24	84%	10.73	10.46	−3%
aes	79	79	64	23	71%	0.82	0.81	−1%
linq_dictionary	102	102	102	66	35%	81.00	79.35	−2%
linq_aggregate	66	66	33	31	53%	308.29	314.23	2%
linq_functional	101	101	88	87	14%	530.94	536.41	1%
linq_enumerable	17	17	17	17	0%	85.55	84.49	−1%
linq_action	11	11	11	11	0%	21.61	21.99	2%
opensrc	**524**	**524**	**444**	**259**	**51%**	**148.42**	**149.68**	**−1%**
lzma–full	12	12	9	5	58%	0.39	0.39	0%
life	4	4	4	4	0%	1.53	1.61	5%
fannkuch	4	4	4	4	0%	1.97	2.01	2%
helloworld	4	4	4	4	0%	1.46	1.47	1%
helloworld–fast	4	4	4	4	0%	1.44	1.38	−4%
hashtest	4	4	4	4	0%	3.66	3.73	2%
fasta	4	4	4	4	0%	3.21	3.50	9%
emscripten	**36**	**36**	**33**	**29**	**19%**	**1.95**	**2.01**	**3%**

Table 2: Table summarizing the precision and performance benefits of various type refinement optimizations. For each benchmark, we provide the number of type-errors reported by the analysis under the four configurations (**B**, **T**, **TP** and **TPUNF**—columns 2–5), the percentage reduction in number of type errors reported when run with **TPUNF** version over the **B** version (column 6), the mean runtime in seconds when run with **B** and **TPUNF** versions (columns 7 and 8, respectively), and the percentage increase in time taken of **TPUNF** version over **B** version (column 9). We also summarize for each benchmark category, the total number of error reports across benchmarks in that category for each version of the analysis, and the mean runtime for running the analysis under **B** and **TPUNF** versions across the benchmarks in that category. The mean performance data has a relative standard deviation of at most 30%.

ment for implicit conditionals in JavaScript can have a significant impact on precision (upto 86% for a static type-error client). We also show that type refinement does not cause any adverse performance impact.

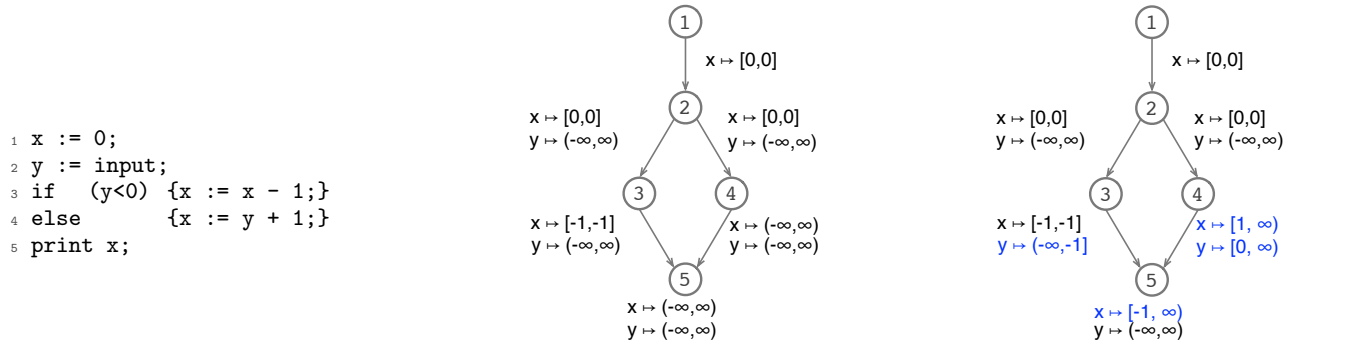
We can do type refinements only if the refinement can be a strong update, thus, we can increase the precision due to type refinement by implementing orthogonal techniques that can increase the number of strong updates. We are currently exploring recency abstraction [10, 22], which is a technique that can increase the number of strong updates in the analysis. It would be interesting to study the effect of combining type refinement with recency abstraction.

We would also like to extend our type refinement to more complicate conditionals, and explore how far we can increase precision benefits without affecting adversely performance.

A. Background on Analysis and Refinement

This section provides a high-level overview of program analysis for those who may be unfamiliar with its terminology and tradeoffs. It also describes a general notion of *refinement* in static analysis. A reader who is familiar with the standard frameworks of dataflow analysis [25] or abstract interpretation [12] should feel free to skip this section.

Static program analysis computes invariants about a program's behavior: things that are guaranteed to *always* happen in every execution, or *never* happen in any execution. Program invariants are used in tasks such as verification, error-checking, and optimization, among other applications. The problem faced by static analysis is that, in general, computing exact invariants about programs is undecidable, as shown by Rice's Theorem [29]. Instead, a program analysis must *approximate* the set of true program invariants. Therefore, an analysis designer must juggle *soundness* (any reported invariant is actually an invariant); *precision* (the analysis computes a close approximation to the set of real program invariants); and *tractability* (the analysis computes its result using a reasonable amount of resources).



(a) An example program that operates over integers. An invariant of this program is that, at line 5, $(\texttt{x} = -1) \vee (\texttt{x} \geq 1)$.

(b) A standard, interval-based analysis of the example program cannot discover any useful invariants about x.

(c) Refinement discovers the conservative invariant that, at line 5, $\texttt{x} \geq -1$. The blue values highlight the benefits of refinement.

Figure 6: An example program and an analysis over its control-flow graph. The analysis uses the interval abstract domain.

A major source of approximation is the method chosen by the analysis designer to represent the set of possible program values. Concretely, i.e., when the program executes, the set of possible values may be infinite (for example, the set of all possible integers); trying to exactly represent this set of *concrete values* makes the analysis undecidable. The analysis designer must instead choose a tractable set of *abstract values* with which to abstractly compute the analysis; this set is called the *abstract domain*. Each value of the abstract domain represents a specific set of concrete values. For the set of integers, one possible abstract domain is $\{+, -, 0\}$, where $+$ represents all positive integers, $-$ represents all negative integers, and 0 represents the single integer 0. This is a very course approximation of integers. A more precise approximation would be the interval abstract domain, where the interval value $[a, b]$ represents all the integers from a to b, inclusive. The least precise interval value is $(-\infty, \infty)$; the most precise interval value is $[a, b]$ where $a = b$.

The choice of abstract domain has a large part in determining the precision of the static analysis. However, even for a given abstract domain the analysis precision can vary based on a number of factors. We illustrate these concepts and tradeoffs by way of example. Figure 6a contains a program in a simple language. In this language all program values are integers. An analysis designer may choose to approximate these integers using intervals. The interval abstract domain might be useful, e.g., as the basis for an analysis that determines which JavaScript values can be safely represented as a 32-bit integer instead of the default (and less efficient) 64-bit floating-point representation [27]. We will first discuss a basic version of the analysis that does not use refinement, and then show how refinement can improve precision while still using the same abstract domain of intervals.

A.1 Basic Analysis

A program's control-flow is approximated as a graph, where the nodes are program statements and there is an edge from s to s' if statement s' can execute immediately after statement s. Figure 6b contains the control-flow graph for our example program.

Given an abstract domain and an acyclic control-flow graph, the analysis algorithm is fairly straightforward.[4] If a node s has a single predecessor s', the algorithm abstractly interprets the statement s on its input, meaning that it performs the same operation as the actual program execution would perform, except it does so on abstract values instead of actual program values. Then it sends the result along the successor edges of s. For example, the analysis interprets the constant 0 in line 1 of the example program as the abstract value $[0, 0]$. The analysis captures the fact that x is bound to $[0, 0]$ (denoted, $x \mapsto [0, 0]$) and sends this information along the edge to statement 2. The analysis interprets the expression input in line 2 as $(-\infty, \infty)$ because the user might input any integer. The analysis then binds this value to y and sends the updated bindings along the successor edges.

If a node has multiple successors, then that node represents a branch. The analysis interprets the branch condition, to decide which branch statement(s) to interpret next, i.e., the true or false branch. Statically determining which branch statements will execute is undecidable, in general. Instead, the analysis conservatively interprets any branch statement that *may* execute, under the abstract interpretation of the branch condition. If the analysis has sufficient information to determine that exactly one branch may execute, then the choice is said to be *deterministic*. If the analysis cannot determine that exactly one branch may execute, then the choice is said to be *non-deterministic* in the sense that the analysis executes *both* branches.

In our example program, after line 2, y has the value $(-\infty, \infty)$. This interval covers both negative and non-negative values, so the branch condition in line 3 leads to a non-deterministic choice. The analysis conservatively over-approximates the program's behavior by copying the values along the edges that correspond to both branches of the if statement. The analysis then interprets those statements and passes the updated bindings along to their respective successors.

If a node has multiple predecessors, then it represents a merge. The analysis first merges all of the inputs from the predecessor edges, then abstractly interprets the node with respect to the merged inputs and sends the results along the node's output(s). For the domain of intervals, the merge of two intervals $[a_{min}, a_{max}]$ and $[b_{min}, b_{max}]$ is $[min(a_{min}, b_{min}), max(a_{max}, b_{max})]$.

This analysis, conducted on our example program, concludes that both x and y can have the value $(-\infty, \infty)$ at line 5. This imprecise conclusion fails to capture an invariant of the program:

[4] The general algorithm, which operates over a possibly cyclic graph, is more complex, but the complexity is not needed to understand refinement. We refer the interested reader to the full frameworks based on fixed point approximation [12, 25].

`x` can never be less than -1. The imprecision results from the fact that the analysis does not interpret the branch condition. The condition is a message from the programmer: `x`'s value depends on `y`'s only when $y \geq 0$. This observation is at the heart of *refinement*: sometimes, it may be possible to interpret a branch condition and to filter the values that flow to the branches.

A.2 Refinement

Refinement improves the precision of our example analysis. The branch condition on line `3` is simple. It can be encoded as a filter for the abstract values flowing through that condition: In the true branch `y`'s abstract value must be such that the branch condition would evaluate to true, and in the false branch `y`'s abstract value must be such that the branch condition would evaluate to false. In our example, this reasoning allows the analysis to determine that `y` must have the abstract value $(-\infty, -1]$ in the true branch and the abstract value $[0, \infty)$ in the false branch, as opposed to having the abstract value $(-\infty, \infty)$ in both branches as in the basic analysis. This increase in precision eventually enables the analysis to discover the invariant at line `5` that $x \geq -1$. (This information is still imprecise; it misses the invariant that `x` can never be 0. However, this over-approximation is an artifact of the abstract domain, which can encode only continuous ranges of values. A more precise abstract domain would be required to capture discontinuities.)

Acknowledgments

We thank Kyle Dewey, Ethan A. Kuefner, and Kevin Gibbons for their help in developing the static analysis infrastructure and its formal description. This work was supported by NSF CCF-1117165.

References

[1] Defensive JavaScript. `http://www.defensivejs.com/`. Accessed: 2013-06-05.

[2] Emscripten. `http://emscripten.org/`. Accessed: 2013-06-05.

[3] LINQ for JavaScript. `http://linqjs.codeplex.com/`. Accessed: 2013-06-05.

[4] Octane JavaScript Benchmark. `http://code.google.com/p/octane-benchmark/`. Accessed: 2013-06-05.

[5] Rhino Documentation. `https://developer.mozilla.org/en-US/docs/Rhino`. Accessed: 2013-06-05.

[6] SunSpider JavaScript Benchmark. `http://www.webkit.org/perf/sunspider/sunspider.html`. Accessed: 2013-06-05.

[7] T.J. Watson Libraries for Analysis (WALA). `http://wala.sf.net`. Accessed: 2013-06-05.

[8] J.-h. D. An, A. Chaudhuri, J. S. Foster, and M. Hicks. Dynamic inference of static types for ruby. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2011.

[9] C. Anderson, P. Giannini, and S. Drossopoulou. Towards type inference for JavaScript. In *European Conference on Object-Oriented Programming (ECOOP)*, 2005.

[10] G. Balakrishnan and T. Reps. Recency-abstraction for heap-allocated storage. In *Symposium on Static Analysis (SAS)*, 2006.

[11] R. Chugh, P. M. Rondon, and R. Jhala. Nested refinements: a logic for duck typing. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2012.

[12] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 1977.

[13] A. Feldthaus, T. D. Millstein, A. Møller, M. Schäfer, and F. Tip. Tool-supported refactoring for JavaScript. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2011.

[14] T. Freeman and F. Pfenning. Refinement types for ML. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1991.

[15] M. Furr, J.-h. D. An, J. S. Foster, and M. Hicks. Static type inference for ruby. In *ACM symposium on Applied Computing*, 2009.

[16] M. Gorbovitski, Y. A. Liu, S. D. Stoller, T. Rothamel, and T. K. Tekle. Alias analysis for optimization of dynamic languages. In *Symposium on Dynamic Languages (DLS)*, 2010.

[17] S. Guarnieri and B. Livshits. GATEKEEPER: mostly static enforcement of security and reliability policies for JavaScript code. In *USENIX security symposium (SSYM)*, 2009.

[18] S. Guarnieri, M. Pistoia, O. Tripp, J. Dolby, S. Teilhet, and R. Berg. Saving the world wide web from vulnerable javascript. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2011.

[19] A. Guha, S. Krishnamurthi, and T. Jim. Static analysis for ajax intrusion detection. In *International World Wide Web Conference (WWW)*, 2009.

[20] A. Guha, C. Saftoiu, and S. Krishnamurthi. Typing local control and state using flow analysis. In *European Symposium on Programming (ESOP)*, 2011.

[21] B. Hackett and S. Guo. Fast and precise hybrid type inference for javascript. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2012.

[22] P. Heidegger and P. Thiemann. Recency Types for Analyzing Scripting Languages. In *European Conference on Object-Oriented Programming*, 2010.

[23] D. Jang and K.-M. Choe. Points-to analysis for javascript. In *ACM symposium on Applied Computing (SAC)*, 2009.

[24] S. H. Jensen, A. Møller, and P. Thiemann. Type Analysis for JavaScript. In *Symposium on Static Analysis (SAS)*, 2009.

[25] G. A. Kildall. A unified approach to global program optimization. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 1973.

[26] H. Lee, S. Won, J. Jin, J. Cho, and S. Ryu. Safe: Formal specification and implementation of a scalable analysis framework for ecmascript. In *International Workshop on Foundations of Object-Oriented Languages (FOOL)*, 2012.

[27] F. Logozzo and H. Venter. RATA: rapid atomic type analysis by abstract interpretation – application to JavaScript optimization. In *Joint European conference on Theory and Practice of Software, international conference on Compiler Construction (CC/ETAPS)*, 2010.

[28] F. Pluquet, A. Marot, and R. Wuyts. Fast type reconstruction for dynamically typed programming languages. In *Symposium on Dynamic Languages (DLS)*, 2009.

[29] H. G. Rice. Classes of Recursively Enumerable Sets and Their Decision Problems. *Transactions of the American Mathematical Society*, 74(2), 1953.

[30] M. Sridharan, J. Dolby, S. Chandra, M. Schäfer, and F. Tip. Correlation tracking for points-to analysis of javascript. In *European Conference on Object-Oriented Programming (ECOOP)*, 2012.

[31] S. Tobin-Hochstadt and M. Felleisen. The design and implementation of typed scheme. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2008.

[32] S. Tobin-Hochstadt and M. Felleisen. Logical types for untyped languages. In *ACM SIGPLAN International Conference on Functional programming (ICFP)*, 2010.

[33] T. Zhao. Polymorphic type inference for scripting languages with object extensions. In *Symposium on Dynamic Languages (DLS)*, 2011.