Static Analysis with Demand-Driven Value Refinement

BENNO STEIN, University of Colorado Boulder, USA
BENJAMIN BARSLEV NIELSEN, Aarhus University, Denmark
BOR-YUH EVAN CHANG, University of Colorado Boulder, USA
ANDERS MØLLER, Aarhus University, Denmark

Static analysis tools for JavaScript must strike a delicate balance, achieving the level of precision required by the most complex features of target programs without incurring prohibitively high analysis time. For example, reasoning about dynamic property accesses sometimes requires precise relational information connecting the object, the dynamically-computed property name, and the property value. Even a minor precision loss at such critical program locations can result in a proliferation of spurious dataflow that renders the analysis results useless.

We present a technique by which a conventional non-relational static dataflow analysis can be combined soundly with a value refinement mechanism to increase precision on demand at critical locations. Crucially, our technique is able to incorporate relational information from the value refinement mechanism into the non-relational domain of the dataflow analysis.

We demonstrate the feasibility of this approach by extending an existing JavaScript static analysis with a demand-driven value refinement mechanism that relies on backwards abstract interpretation. Our evaluation finds that precise analysis of widely used JavaScript utility libraries depends heavily on the precision at a small number of critical locations that can be identified heuristically, and that backwards abstract interpretation is an effective mechanism to provide that precision on demand.

CCS Concepts: • **Theory of computation → Program analysis**.

Additional Key Words and Phrases: JavaScript, dataflow analysis, abstract interpretation

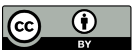
ACM Reference Format:
Benno Stein, Benjamin Barslev Nielsen, Bor-Yuh Evan Chang, and Anders Møller. 2019. Static Analysis with Demand-Driven Value Refinement. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 140 (October 2019), 29 pages. https://doi.org/10.1145/3360566

1 INTRODUCTION

Although the many dynamic features of the JavaScript programming language provide great flexibility, they also make it difficult to reason statically about dataflow and control-flow. Several research tools, including TAJS [Jensen et al. 2009], WALA [Sridharan et al. 2012], SAFE [Lee et al. 2012], and JSAI [Kashyap et al. 2014], have been developed in recent years to address this challenge. A notable trend is that analysis precision is being increased in many directions, including high degrees of context sensitivity [Andreasen and Møller 2014], aggressive loop unrolling [Park and Ryu 2015], and sophisticated abstract domains for strings [Amadini et al. 2017; Madsen and Andreasen 2014; Park et al. 2016], to enable analysis of real-world JavaScript programs.

The need for precision in analyzing JavaScript is different from other programming languages. For example, it is widely recognized that, for Java analysis, choosing between different degrees

Authors' email addresses: benno.stein@colorado.edu, barslev@cs.au.dk, evan.chang@colorado.edu, amoeller@cs.au.dk.



This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2019 Copyright held by the owner/author(s).
2475-1421/2019/10-ART140
https://doi.org/10.1145/3360566

of context sensitivity is a trade-off between analysis precision and performance. With JavaScript, the relationship between precision and performance is more complicated: low precision tends to cause an avalanche of spurious dataflow, which slows down the analysis and often renders it useless [Andreasen and Møller 2014; Sridharan et al. 2012].

Unfortunately, uniformly increasing analysis precision to accommodate the patterns found in JavaScript programs is not a viable solution, because the high precision that is critical for some parts of the programs may be overkill for others. For example, the approach taken by SAFE performs loop unrolling indiscriminately whenever the loop condition is determinate [Park and Ryu 2015], which is often unnecessary and may be costly. Another line of research attempts to address this problem by identifying specific syntactic patterns known to be particularly difficult to analyze and applying variations of trace partitioning to handle those patterns more precisely [Ko et al. 2017, 2019; Sridharan et al. 2012].

In this work, we explore a different idea: instead of pursuing ever more elaborate abstract domains, context-sensitivity policies, or syntactic special-cases, we augment an existing static dataflow analysis by a novel *demand-driven value refinement* mechanism that can eliminate spurious dataflow at critical places with a targeted backwards analysis. Our approach is inspired by Blackshear et al. [2013], who introduced the use of a backwards analysis to identify spurious memory leak alarms produced by a Java points-to analysis. We extend their technique by applying the backwards analysis *on-the-fly*, to avoid critical precision losses during the main dataflow analysis, rather than as a post-processing alarm triage tool. Also, our technique is designed to handle the dynamic features of JavaScript that do not appear in Java.

We find that demand-driven value refinement is particularly effective for providing precise *relational* information, even though the abstract domain of the underlying dataflow analysis is non-relational. Such relational information is essential for the precise analysis of many common dynamic language programming paradigms, especially those found in widely-used libraries like Underscore[1] and Lodash[2] that rely heavily on metaprogramming.

An important observation that enables our approach is that the extra precision is typically only critical at very few program locations, and that these locations can be identified during the main dataflow analysis by inspecting the abstract values it produces.

In summary, the contributions of this paper are as follows.

- We present the idea of demand-driven value refinement as a technique for soundly eliminating critical precision losses in static analysis for dynamic languages (Section 4). For clarity, the presentation is based on a dataflow analysis framework for a minimal dynamic programming language (Section 3).
- We present a separation logic-based backwards abstract interpreter, which can answer value refinement queries to precisely refine abstract values and provide relational precision to the non-relational dataflow analysis as an abstract domain reduction. This backwards analysis is first described for the minimal dynamic language (Section 5) and then for JavaScript (Section 6).
- We empirically evaluate our technique using an implementation, TAJS_{VR}, for JavaScript (Section 7). We find that demand-driven value refinement can provide the necessary precision to analyze code in libraries that no existing static analysis is able to handle. For example, the technique enables precise analysis of 266 of 306 test cases from the latest version of Lodash (the most depended-upon package in the npm repository), all of which are beyond the reach of other state-of-the-art analyses.

[1] https://underscorejs.org/
[2] https://lodash.com/

```
1  function mixin(object, source) {
2      var methodNames = baseFunctions(source, Object.keys(source));
3      arrayEach(methodNames, function(methodName) {
4          var func = source[methodName];
5          object[methodName] = func;
6          if (isFunction(object)) {
7              object.prototype[methodName] = function() {
8                  ...
9                  return func.apply(...);
10             }
11         }
12     })
13 }
```

(a) Lodash's library function `mixin` (simplified for brevity).

```
14 function baseFor(object, iteratee) {
15     var index = -1,
16     props = Object.keys(object),
17     length = props.length;
18     while (length--) {
19         var key = props[++index];
20         iteratee(object[key], key)
21     }
22 }
23
24 mixin(lodash, (function() {
25     var source = {};
26     baseFor(lodash, function(func, methodName) {
27         if (!hasOwnProperty.call(lodash.prototype, methodName)) {
28             source[methodName] = func;
29         }
30     });
31     return source;
32 }()));
```

(b) A use of `mixin` in Lodash's bootstrapping.

Fig. 1. Excerpts from the Lodash library. Dynamic property writes that require relational information to analyze precisely are highlighted by arrows connecting them to corresponding dynamic property reads.

2 MOTIVATING EXAMPLE

The example program in Fig. 1 is an excerpt from Lodash 4.17.10. It consists of a library function `mixin` (Fig. 1a) and a simple use of `mixin` (Fig. 1b) originating from the bootstrapping of the library. The `mixin` function copies all methods from the `source` parameter into the `object` parameter. If `object` is a function, the methods are also written to the prototype of `object`, such that instantiations of `object` (using the keyword `new`) also have the methods. The function `baseFor` invokes the `iteratee` function on each property of the given object. The `mixin` function is called in line 24, where the first argument is the `lodash` library object and the second argument is an object, created using `baseFor`, that consists of all properties of `lodash` that are not already in `lodash.prototype`. The purpose of this call to `mixin` is to make each method that is defined on `lodash`, for example `lodash.map`, available on objects created by the `lodash` constructor.

This code contains three dynamic property read/write pairs – indicated by the labels ①, ②, and ③ in Fig. 1 – where relational information connecting the property name of the read and write is essential to avoid crippling loss of precision.

All three labelled read/write pairs are instances of a metaprogramming pattern called *field copy or transformation (FCT)* [Ko et al. 2017, 2019], which consists of a property read operation `x[a]` and a property write operation `y[b] = v` where the property name `b` is a function of the property name `a` and the written value `v` is a function of the read value `x[a]`. This pattern is a generalization of the *correlated read/write* pattern [Sridharan et al. 2012], which requires that the property names are strictly equal. Our technique attempts to generalize such syntactic patterns by identifying imprecise dynamic property writes *semantically* during the dataflow analysis. The benefits of this semantic approach are twofold: it avoids the brittleness of syntactic patterns, and it only incurs additional analysis cost where needed, rather than at all locations matching a syntactic pattern.

Analysis precision at dynamic property read and write operations is known to be critical for static analysis for JavaScript programs [Andreasen and Møller 2014; Ko et al. 2017, 2019; Sridharan et al. 2012]. If the abstract values of the property names are imprecise, then a naive analysis will mix together the values of the different properties, which often causes a catastrophic loss of precision. In the case of Lodash, such a naive analysis of the bootstrapping code would essentially cause all the library functions to be mixed together, making it impossible to analyze any realistic applications of the library.

Existing attempts to address this problem are unfortunately insufficient. WALA [Sridharan et al. 2012], SAFE$_{LSA}$ [Park and Ryu 2015], and TAJS [Andreasen and Møller 2014] use context-sensitivity and loop-unrolling to attempt to obtain precise information about the property names, but fail to provide precise values of the variables `methodName` (at lines 4, 5, and 28) and `key` (at line 20).[3]

The CompAbs analyzer [Ko et al. 2017, 2019] takes a different approach that does not require precise tracking of the possible values of `methodName` and `key`. Instead, it attempts to syntactically identify correlated dynamic property reads and writes and applies trace partitioning [Rival and Mauborgne 2007] at the relevant property read operations. However, CompAbs fails to identify any of the three highlighted read/write pairs in Fig. 1 due to the brittleness of syntactic patterns. While it might be possible to detect ② syntactically, the trace partitioning approach is insufficient for that read/write pair, since the value in this case flows through a free variable (`func`) that is shared across all partitions. As a result, CompAbs fails to analyze the full Lodash library with sufficient precision and ends up mixing together all properties of the `lodash` object.

Triggering Demand-Driven Value Refinement. Our approach is able to achieve sufficient precision for all three dynamic property read/write pairs in the example without relying on brittle syntactic patterns to identify such pairs.

The key idea underlying our technique is to detect *semantically* when an imprecise property write is about to occur during the dataflow analysis, at which point we apply a targeted value refinement mechanism to recover the relational information needed to precisely determine which values are written to which heap locations. More specifically, when the analysis encounters a dynamic property write `obj[p] = v` and has imprecise abstract values for `p` and `v`, we decompose the abstract value of `v` into a set of more precise partitions and then query the refinement mechanism to determine, for each partition, the possible values of `p`. Now, instead of writing the imprecise value of `v` to all the property names of `obj` that match the imprecise value of `p`, we write each partition of `v`

[3]For example, achieving sufficient precision for `methodName` at lines 4 and 5 requires that the analysis can infer the precise length and contents of the `methodNames` array, which is beyond the capabilities of those analyzers, even if using an (unsound) assumption that the order of the entries of the array returned by `Object.keys` is known statically.

only to the corresponding refined property names of `obj`, thereby recovering relational information between `p` and `v`.

For example, suppose that the dataflow analysis reaches the dynamic property write of ③ (at line 28) with an abstract state mapping the property name `methodName` to the abstract value denoting any string and the value `func` to the abstract value that abstracts all functions in `lodash`. We then decompose `func` into precise partitions – one for each of the functions – and query the value refinement mechanism for the corresponding sets of possible property names. Recovering that relational information, we obtain a unique function for each property name, such that the `lodash.map` function is assigned to `source["map"]`, `lodash.filter` is assigned to `source["filter"]`, etc., instead of mixing them all together. This technique handles case ① analogously, by detecting the imprecision semantically at the dynamic property write.

For property read/write pair ②, however, the value to be written is a precise function (the anonymous function in lines 7–10) and the imprecise value `func` is a free variable declared in an enclosing function. In this case, value refinement is not triggered until `func` is called on line 9, at which point we apply the same value refinement technique as above and recover the necessary relational information to precisely resolve the target of that call, as described in further detail in Section 6.2.

Unlike abstraction-refinement techniques (see Section 8), this mechanism is able to recover relational information and use it to regain precision in the dataflow analysis without any modifications to its non-relational abstract domain and without restarting the entire analysis.

Value Refinement using Backwards Analysis. Our value refinement mechanism is powered by a goal-directed backwards abstract interpreter. Given a program location ℓ, a program variable y, and a constraint ϕ, it computes a bound on the possible values of y at ℓ in concrete states satisfying ϕ. We refer to the forward dataflow analysis as the *base analysis* and the backwards analysis as the *value refiner*.

For example, if asked to refine the variable `methodName` at the location preceding line 28, under the condition that `func` is the `lodash.map` function, our value refiner can determine that `methodName` must be `"map"`. In doing so, the value refiner provides targeted information about the relation between `func` and `methodName` to the base analysis.

Intuitively, the value refiner works by overapproximately traversing the abstract state space backwards from the given program location, accumulating symbolic constraints about the paths leading to that location. The traversal proceeds along each such path until a sufficiently precise value is obtained for the desired program variable. In this process, the value refiner takes advantage of the current call graph and the abstract states computed so far by the base analysis. The resulting abstract value thereby overapproximates the possible values of y at ℓ, for all program executions where ϕ is satisfied at ℓ and that are possible according to the call graph and abstract states from the base analysis. As we argue in Sections 4 and 7, the value refinement mechanism is sound even though it relies on information from the base analysis that has not yet reached its fixpoint.

3 A SIMPLE DYNAMIC LANGUAGE AND DATAFLOW ANALYSIS

To provide a foundation for explaining our demand-driven value refinement mechanism, in Section 3.1 we define the syntax and concrete semantics of a small dynamic language. This language is designed for simplicity and clarity of presentation and is meant to illustrate some of the core challenges in dynamic language analysis without the complexity that arises from a tested core calculus for JavaScript such as λ_{JS} [Guha et al. 2010]. We then define our analysis over this minimal language in Section 3.2 and describe the extensions needed to handle the full JavaScript language in Section 6.

$$
\begin{array}{rll}
\text{variables} & x, y, z \in \mathit{Var} & \\
\text{primitives} & p \in \mathit{Prim} & ::= \texttt{undef} \mid \texttt{true} \mid \texttt{false} \\
& & \quad \mid \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \ldots \\
& & \quad \mid \texttt{"foo"} \mid \texttt{"bar"} \mid \ldots \\
\text{statements} & s \in \mathit{Stmt} & ::= x\texttt{=\{\}} \mid x\texttt{=}y \mid x\texttt{=}p \\
& & \quad \mid x\texttt{=}y \oplus z \mid \texttt{assume}\ x \\
& & \quad \mid x\texttt{[}y\texttt{]=}z \mid x\texttt{=}y\texttt{[}z\texttt{]} \\
\text{operators} & \oplus & ::= + \mid - \mid = \mid \neq \mid \ldots \\
\text{locations} & \ell \in \mathit{Loc} & \\
\text{control edges} & t \in \mathit{Trans} & ::= \ell \rightarrow_s \ell'
\end{array}
$$

Fig. 2. Concrete syntax for a simple dynamic language.

$$
\begin{array}{rll}
\text{object addresses} & a \in \mathit{Addr} & \\
\text{memory addresses} & m \in \mathit{Mem} & = \mathit{Var} \cup (\mathit{Addr} \times \mathit{Prim}) \\
\text{values} & v \in \mathit{Val} & = \mathit{Addr} \cup \mathit{Prim} \\
\text{states} & \sigma \in \mathit{State} & = \mathit{Mem} \hookrightarrow \mathit{Val}
\end{array}
$$

$$
\begin{array}{rl}
[\![\cdot]\!] & : \mathit{Stmt} \rightarrow \mathit{State} \hookrightarrow \mathit{State} \\
[\![x\texttt{=\{\}}]\!](\sigma) & = \sigma[x \mapsto \mathsf{fresh}(\sigma)] \\
[\![x\texttt{=}y]\!](\sigma) & = \sigma[x \mapsto \sigma y] \\
[\![x\texttt{=}p]\!](\sigma) & = \sigma[x \mapsto p] \\
[\![x\texttt{=}y \oplus z]\!](\sigma) & = \sigma[x \mapsto \sigma y \oplus \sigma z] \\
[\![\texttt{assume}\ x]\!](\sigma) & = \sigma \quad \text{if } \sigma x = \texttt{true} \\
[\![x\texttt{[}y\texttt{]=}z]\!](\sigma) & = \sigma[(\sigma x, \sigma y) \mapsto \sigma z] \quad \text{if } \sigma x \in \mathit{Addr} \text{ and } \sigma y \in \mathit{Prim} \\
[\![x\texttt{=}y\texttt{[}z\texttt{]}]\!](\sigma) & = \sigma[x \mapsto \sigma(\sigma y, \sigma z)]
\end{array}
$$

Fig. 3. Denotational semantics and concrete domains.

3.1 A Simple Dynamic Language

The syntax and denotational semantics of our core dynamic language are shown in Fig. 2 and Fig. 3.

A program in this language is an unstructured control-flow graph represented as a pair $\langle \ell_0, T \rangle$ of an initial location $\ell_0 \in \mathit{Loc}$ and a set of control-flow edges $T \subseteq \mathit{Trans}$. A program location $\ell \in \mathit{Loc}$ is a unique identifier. A memory address $m \in \mathit{Mem}$ is either a program variable x or an object property (a, p) where a is the address of an object and p is a primitive value. Concrete states $\sigma \in \mathit{State}$ are partial functions from memory addresses to values, which are either object addresses or primitives.

We write ε for the empty state and use the notation $\sigma[m \mapsto v]$ to denote a state identical to σ except at location m where v is now stored, and σm to denote the value stored at m in σ or `undef` if

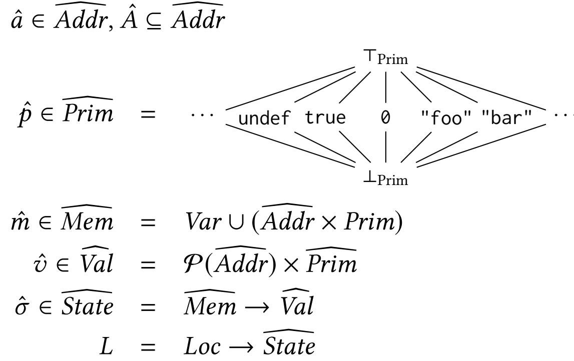


Fig. 4. Dataflow analysis lattice.

no such value exists. We also assume a helper function fresh(σ) that returns an object address that is fresh with respect to state σ.

The denotation of a statement s is a partial function $[\![s]\!]$ from states to states. The collecting semantics of a program $\langle \ell_0, T \rangle$ is defined in terms of the denotational semantics as a function $[\![_]\!]_{\langle \ell_0, T \rangle} : Loc \rightarrow \mathcal{P}(State)$ that captures the reachable state space of the program, as the least solution to the following constraints:

$$\varepsilon \in [\![\ell_0]\!]_{\langle \ell_0, T \rangle}$$

$$\forall \sigma \in [\![\ell]\!]_{\langle \ell_0, T \rangle} \text{ and } \ell \rightarrow_s \ell' \in T : [\![s]\!](\sigma) \in [\![\ell']\!]_{\langle \ell_0, T \rangle}$$

The first constraint says that the empty state ε is reachable at the initial location. The second constraint defines the successor states according to the denotational semantics.

3.2 Dataflow Analysis

We now describe a basic dataflow analysis for this minimal dynamic language, which we will extend in Section 4 with support for demand-driven value refinement.

The analysis is expressed as a monotone framework [Kam and Ullman 1977] consisting of a domain of abstract states and monotone transfer functions for the different kinds of statements. Programs can then be analyzed by a fixpoint solver computing an overapproximate abstract state for each program location.[4]

The analysis domain we use is the lattice L described in Fig. 4, which is a simplified version of the one used by TAJS [Jensen et al. 2009]. For each program location, an element of L provides an abstract state, which maps abstract memory addresses to abstract values. Object addresses are abstracted using, for example, allocation-site abstraction [Chase et al. 1990]. The domain of abstract values, $\widehat{Val}$, is a product of two sub-domains describing references to objects and primitive values, respectively, using the constant propagation lattice to model the latter.

We write $\hat{a} \prec_1 \hat{v}$ if the first component of the abstract value $\hat{v}$ contains the abstract object address $\hat{a}$, and similarly, $\hat{p} \prec_2 \hat{v}$ means that the concretization of the abstract primitive value $\hat{p}$ is a subset of the concretization of $\hat{v}$. We use $\hat{v}_1 \sqcup \hat{v}_2$ to denote the least upper bound of $\hat{v}_1$ and $\hat{v}_2$.

The semantics of each control edge $\ell \rightarrow_s \ell'$ is modeled abstractly by the transfer function $\mathcal{T}_{\ell \rightarrow_s \ell'} : \widehat{State} \rightarrow \widehat{State}$. When the statement s is a dynamic property write `x[y]=z`, the transfer

[4] We assume the reader is familiar with the basic concepts of abstract interpretation [Cousot and Cousot 1992], including the terms *abstraction*, *concretization*, *collecting semantics*, and *soundness*.

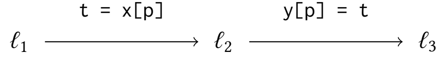


Fig. 5. A program fragment with a correlated property read/write pair.

function is defined as follows:

$$\mathcal{T}_{\ell \to_s \ell'}(\hat{\sigma})(\hat{m}) = \begin{cases} \hat{\sigma}\hat{m} \sqcup \hat{\sigma} z & \text{if } \hat{m} = (\hat{a}, p) \ \wedge \ \hat{a} \prec_1 \hat{\sigma} x \ \wedge \ p \prec_2 \hat{\sigma} y \\ \hat{\sigma}\hat{m} & \text{otherwise} \end{cases}$$

In other words, the analysis models such an operation by weakly[5] updating all the affected abstract memory addresses with the abstract value of z. Due to the limited space, we omit descriptions of the remaining analysis transfer functions for other kinds of statements; it suffices to require that they soundly overapproximate the semantics [Cousot and Cousot 1977].

The refinement mechanism directly involves the fixpoint solver. As customary in dataflow analysis, we assume the fixpoint solver uses a worklist algorithm to determine which locations to process next each time a transfer function has been applied [Kildall 1973]. A worklist algorithm relies on a map $dep : Loc \to \mathcal{P}(Loc)$, such that $dep(\ell)$ contains all direct dependents of ℓ, in our case the successors $\{\ell' \mid \ell \to_s \ell' \in Trans\}$. For a worklist algorithm to be sound, it must add all the locations $dep(\ell)$ to the worklist when the abstract state at ℓ is updated.

Example. Fig. 5 shows a program fragment with a correlated property read/write pair like in Section 2. Assume $\hat{\sigma}$ is an abstract state where `x` and `y` point to distinct objects ($\hat{a}_{\mathtt{x}}$ and $\hat{a}_{\mathtt{y}}$, respectively), `p` is any primitive value, the `x` object has three properties (named `"a"`, `"b"`, and `"c"`) with different values, and the `y` object is empty:

$$\hat{\sigma}\mathtt{x} = (\{\hat{a}_{\mathtt{x}}\}, \bot_{\mathrm{Prim}}) \qquad \hat{\sigma}(\hat{a}_{\mathtt{x}}, \mathtt{"a"}) = (\{\hat{a}_{\mathtt{xa}}\}, \bot_{\mathrm{Prim}})$$
$$\hat{\sigma}\mathtt{y} = (\{\hat{a}_{\mathtt{y}}\}, \bot_{\mathrm{Prim}}) \qquad \hat{\sigma}(\hat{a}_{\mathtt{x}}, \mathtt{"b"}) = (\{\hat{a}_{\mathtt{xb}}\}, \bot_{\mathrm{Prim}})$$
$$\hat{\sigma}\mathtt{p} = (\emptyset, \top_{\mathrm{Prim}}) \qquad \hat{\sigma}(\hat{a}_{\mathtt{x}}, \mathtt{"c"}) = (\emptyset, \top_{\mathrm{Prim}})$$
$$\hat{\sigma}(\hat{a}_{\mathtt{y}}, p) = (\emptyset, \bot_{\mathrm{Prim}}) \quad \text{for all } p \in Prim$$

The transfer function for `t = x[p]` with this abstract state at initial program location ℓ_1 yields an abstract state at ℓ_2 that maps `t` to $(\{\hat{a}_{\mathtt{xa}}, \hat{a}_{\mathtt{xb}}\}, \top_{\mathrm{Prim}})$. Next, the transfer function for `y[p] = t` as defined above results in an abstract state at ℓ_3 where every property of the abstract object $\hat{a}_{\mathtt{y}}$ has the same abstract value as `t`, meaning that they all may point to any of the two abstract objects $\hat{a}_{\mathtt{xa}}$ and $\hat{a}_{\mathtt{xb}}$ and have any primitive value. Consequently, the analysis result is very imprecise. In the following section, we explain how the basic dataflow analysis can be extended with demand-driven value refinement to avoid the precision loss.

4 DEMAND-DRIVEN VALUE REFINEMENT

In this section, we introduce the notion of a *value refiner* (Section 4.1), discuss when and how to apply value refinement during the base analysis (Section 4.2), and explain how to integrate a value refiner into the base analysis in a way that allows the value refiner to benefit from the abstract states that are constructed by the base analysis (Section 4.3).

[5] Our implementation for JavaScript uses a more expressive heap abstraction that permits strong updates [Chase et al. 1990; Jensen et al. 2009] in certain situations.

4.1 Value Refinement

A *value refiner* is a function

$$R : Loc \times Var \times Constraint \rightarrow \mathcal{P}(\widehat{Val})$$

that, given a program location $\ell \in Loc$, a program variable $y \in Var$, and a constraint $\phi \in Constraint = Var \times \widehat{Val}$ yields a set of abstract values that are possible for y at ℓ in states that satisfy ϕ. We refer to an invocation of the value refiner function by the base analysis as a *refinement query*. For the refinement queries we need, a constraint is simply a pair of a program variable and an abstract value, written $z \mapsto \hat{v}$, specifying that the variable z has value $\hat{v}$.

We require R to be *sound* in the sense that it overapproximates all possible behaviors of the program according to the collecting semantics: for every state $\sigma \in [\![\ell]\!]_{\langle \ell_0, T\rangle}$ where ℓ is a location in the program $\langle \ell_0, T\rangle$ and the abstraction of σ satisfies the constraint ϕ, the value σy is in the concretization of an abstract value in $R(\ell, y, \phi)$ for any y.

In Section 5 we present a specific value refiner; for the remainder of the current section we can think of the value refiner as a black-box component with the above properties.

4.2 Using Value Refinement in Dataflow Analysis

Value refinement can in principle be invoked whenever the base analysis detects that a potentially critical loss of precision is about to happen, to provide more precise abstract values. As discussed in Section 2, such precision losses often occur in connection with dynamic property writes, so we here focus on that kind of operation. In Section 6.2, we consider value refinement also at variable read operations.

First, we define a helper function $Part : \widehat{Val} \rightarrow \mathcal{P}(\widehat{Val})$ that partitions an abstract value into a set of abstract values, each containing at most one abstract memory address:

$$Part(\hat{A}, \hat{p}) = \{\ (\{\hat{a}\}, \bot_{\text{Prim}}) \mid \hat{a} \in \hat{A}\ \} \cup \begin{cases} \{(\emptyset, \hat{p})\} & \text{if } \hat{p} \neq \bot_{\text{Prim}} \\ \emptyset & \text{otherwise} \end{cases}$$

Continuing the example from Section 3.2, we have:

$$Part(\{\hat{a}_{\mathsf{xa}}, \hat{a}_{\mathsf{xb}}\}, \top_{\text{Prim}}) = \{(\{\hat{a}_{\mathsf{xa}}\}, \bot_{\text{Prim}}), (\{\hat{a}_{\mathsf{xb}}\}, \bot_{\text{Prim}}), (\emptyset, \top_{\text{Prim}})\}$$

We now incorporate value refinement into the base analysis by replacing the ordinary transfer function $\mathcal{T}_{\ell \rightarrow_s \ell'}$ from Section 3.2 by a new transfer function $\mathcal{T}^{\mathsf{VR}}_{\ell \rightarrow_s \ell'}$. The domain of the base analysis remains unchanged (unlike traditional abstraction refinement techniques). The new transfer function is defined as follows when s is a dynamic property write statement $x[y]=z$:

$$\mathcal{T}^{\mathsf{VR}}_{\ell \rightarrow_s \ell'}(\hat{\sigma})(\hat{m}) = \begin{cases} \hat{\sigma}\hat{m} \ \sqcup\ V(\hat{\sigma}, \ell, y, z, p) & \text{if } \top_{\text{Prim}} \prec_2 \hat{\sigma}y \ \wedge\ |Part(\hat{\sigma}z)| > 1 \\ & \wedge\ \hat{m} = (\hat{a}, p) \ \wedge\ \hat{a} \prec_1 \hat{\sigma}x \\ \mathcal{T}_{\ell \rightarrow_s \ell'}(\hat{\sigma})(\hat{m}) & \text{otherwise} \end{cases}$$

where the abstract value being written is

$$V(\hat{\sigma}, \ell, y, z, p) = \bigsqcup \left\{ \hat{z} \in Part(\hat{\sigma}z) \,\middle|\, \exists \hat{y} \in R(\ell, y, z \mapsto \hat{z}) : p \prec_2 \hat{y} \right\}$$

This modified transfer function captures some of the key ideas of our approach, so we carefully explain each part of the definition. The first case of the transfer function definition shows when and how value refinement is used, and the second case simply falls back to the ordinary transfer function. Value refinement is applied when the analysis has an imprecise value for y (i.e., $\top_{\text{Prim}} \prec_2 \hat{\sigma}y$) and an imprecise value for z that is partitioned nontrivially (i.e., $|Part(\hat{\sigma}z)| > 1$), provided that the desired abstract memory address $\hat{m}$ denotes an object property (i.e., $\hat{m} = (\hat{a}, p)$ for some abstract object

address $\hat{a}$ and property name p). The abstract value $V(\hat{\sigma}, \ell, y, z, p)$ being written is then computed by issuing a refinement query $R(\ell, y, z \mapsto \hat{z})$ for each partition $\hat{z}$ of $\hat{\sigma}z$ (i.e., $\hat{z} \in \mathit{Part}(\hat{\sigma}z)$). Each of the resulting abstract values $\hat{y}$ describes a possible value of y under the constraint that z has value $\hat{z}$. In this way, rather than writing the imprecise abstract value $\hat{\sigma}z$ to all properties of $\hat{a}$, the analysis writes each of the more precise abstract values $\hat{z}$ to the corresponding refined property name p that matches $\hat{y}$ (i.e., $p \prec_2 \hat{y}$).

Example. Recall that in the example from Section 3.2, at the dynamic property write `y[p] = t` the base analysis has imprecise abstract values for the property name `p` and for the value `t` being written. More specifically, *Part* partitions the latter into three more precise abstract values as shown above. This means that the condition is satisfied for using value refinement, so the modified transfer function then issues three refinement queries. Using the value refiner that we present in Section 5 yields the following results:

$$R(\ell_2, \mathtt{p}, \mathtt{t} \mapsto (\{\hat{a}_{\mathtt{xa}}\}, \bot_{\mathrm{Prim}})) = \{(\emptyset, \texttt{"a"})\}$$
$$R(\ell_2, \mathtt{p}, \mathtt{t} \mapsto (\{\hat{a}_{\mathtt{xb}}\}, \bot_{\mathrm{Prim}})) = \{(\emptyset, \texttt{"b"})\}$$
$$R(\ell_2, \mathtt{p}, \mathtt{t} \mapsto (\emptyset, \top_{\mathrm{Prim}})) = \{(\emptyset, \texttt{"c"})\}$$

The transfer function then writes each refined abstract value only to the relevant property of $\hat{a}_{\mathtt{y}}$, instead of mixing them all together like the ordinary transfer function. For example, the resulting state maps $(\hat{a}_{\mathtt{y}}, \texttt{"a"})$ to $(\{\hat{a}_{\mathtt{xa}}\}, \bot_{\mathrm{Prim}})$. The base analysis then proceeds with this more precise abstract state.

Notice that the base analysis has only one abstract value per abstract memory address and program location, whereas the value refiner returns a *set* of abstract values at each refinement query. In the example described above, each of the refinement query results contains only one abstract value, but when applying our technique to the examples from Section 2, we benefit from the possibility that R can return multiple abstract values: Some methods of the `lodash` object are accessible via multiple names, for example `lodash.entries` and `lodash.toPairs` are aliases. In this case, querying the value refiner for the possible property names given that the value being written is that specific function, the result can be expressed as the set of the two strings `"entries"` and `"toPairs"` instead of the less precise single abstract value representing all possible strings.

4.3 Using the Base Analysis During Refinements

The value refiner can leverage the base analysis state to allow for more efficient implementation. For the dataflow analysis defined in Section 3.2, the refiner can read partially-computed abstract states; in our JavaScript implementation (see Section 6.1), the refiner also uses the partially-computed call graph. We argue that this is sound even though the base analysis has not yet reached a fixpoint when the refiner is invoked. This extended kind of a value refiner is denoted R_X where $X : \mathit{Loc} \rightarrow \widehat{\mathit{State}}$ is a lattice element of the base analysis.

For the example from Section 3.2, R_X can obtain the value of the variable `x` at ℓ_1 simply by looking up $X(\ell_1)(\mathtt{x})$ from the base analysis, without needing to traverse all the way back to where the value of `x` was created. Similarly, for analysis of JavaScript, using the call graph available from the base analysis allows the value refiner to narrow its exploration when stepping from function entry points to call sites.

For this mechanism to retain analysis soundness, we slightly modify the base analysis. Recall from Section 3.2 that the base analysis relies on a worklist of program locations. Now, whenever a refinement query is triggered by the base analysis at some program location ℓ and the value refiner reads from the abstract state $X(\ell')$, we extend the *dep* map to record that ℓ depends on ℓ'. In this

$$
\begin{array}{rcl}
\text{symbolic variables} & \hat{x}, \hat{y}, \hat{z}, \textsc{res} & \in \widehat{Var} \\
\text{symbolic expressions} & \hat{e} \in \widehat{Expr} & ::= \hat{x} \mid \hat{v} \mid \hat{e}_1 \oplus \hat{e}_2 \\
\text{symbolic stores} & \varphi \in \widehat{Store} & ::= \hat{h} \wedge \pi \mid \varphi_1 \vee \varphi_2 \\
\text{heap constraints} & \hat{h} & ::= \text{true} \mid \text{unalloc}(\hat{x}) \mid x \mapsto \hat{x} \mid \hat{x}_1[\hat{x}_2] \mapsto \hat{x}_3 \mid \hat{h}_1 * \hat{h}_2 \\
\text{pure constraints} & \pi & ::= \text{true} \mid \hat{e} \mid \pi_1 \wedge \pi_2 \\
\text{valuations} & \eta \in Valua & : \widehat{Var} \to Val
\end{array}
$$

(a) Abstractions for value refinement. Recall from Section 3 that $\oplus$ ranges over binary operators, x over program variables, and $\hat{v}$ over abstract values.

$$
\begin{array}{rcl}
\text{eval} & : & (\widehat{Expr} \times Valua) \to \mathcal{P}(Val) \\
\text{eval}(\hat{x}, \eta) & = & \{\eta \hat{x}\} \\
\text{eval}(\hat{v}, \eta) & = & \gamma_{\text{val}}(\hat{v}) \\
\text{eval}(\hat{e}_1 \oplus \hat{e}_2, \eta) & = & \left\{ v_1 \oplus v_2 \;\middle|\; \begin{array}{l} v_1 \in \text{eval}(\hat{e}_1, \eta) \\ \wedge\; v_2 \in \text{eval}(\hat{e}_2, \eta) \end{array} \right\}
\end{array}
$$

(b) Abstract expression evaluation function eval. The notation $\gamma_{\text{val}}(\hat{v})$ refers to the concretization of $\hat{v}$.

$$
\begin{array}{rcl}
\gamma(\hat{h} \wedge \pi) & = & \gamma(\hat{h}) \cap \gamma(\pi) \\
\gamma(\varphi_1 \vee \varphi_2) & = & \gamma(\varphi_1) \cup \gamma(\varphi_2) \\
\gamma(\text{true}) & = & State \times Valua \\
\gamma(\hat{e}) & = & \{(\sigma, \eta) \mid \text{true} \in \text{eval}(\hat{e}, \eta)\} \\
\gamma(\pi_1 \wedge \pi_2) & = & \gamma(\pi_1) \cap \gamma(\pi_2) \\
\gamma(\text{unalloc}(\hat{x})) & = & \{(\sigma, \eta) \mid \forall v : \sigma(\eta(\hat{x}), v) = \text{undef}\} \\
\gamma(x \mapsto \hat{x}) & = & \{(\sigma, \eta) \mid \sigma x = \eta(\hat{x})\} \\
\gamma(\hat{x}_1[\hat{x}_2] \mapsto \hat{x}_3) & = & \{(\sigma, \eta) \mid \sigma(\eta(\hat{x}_1), \eta(\hat{x}_2)) = \eta(\hat{x}_3)\} \\
\gamma(\hat{h}_1 * \hat{h}_2) & = & \left\{ (\sigma_1 \uplus \sigma_2, \eta) \;\middle|\; \begin{array}{l} (\sigma_1, \eta) \in \gamma(\hat{h}_1) \;\wedge\; (\sigma_2, \eta) \in \gamma(\hat{h}_2) \\ \wedge\; \text{dom}(\sigma_1) \cap \text{dom}(\sigma_2) = \emptyset \end{array} \right\}
\end{array}
$$

(c) Concretizations γ for symbolic stores φ, heap constraints $\hat{h}$, and pure constraints π. We denote by $\uplus$ the union of two partial functions with disjoint domains.

Fig. 6. Syntax and concretizations of abstractions used for value refinement.

way, if the abstract state at ℓ' changes later during the base analysis, the result of the invocation of the value refiner is invalidated and eventually recomputed.[6]

As such, the soundness criterion from Section 4.1 for the value refiner needs to be adjusted by weakening the soundness requirement so that the value refiner needs only overapproximate those concrete program behaviors that are abstracted by the current base analysis state. We say that a trace $\ell_0 \to_{s_1} \ell_1 \to_{s_2} \cdots \to_{s_n} \ell_n$ is *abstracted* by a base analysis state $X \in L$ if, for all $k \leq n$, $(\llbracket s_k \rrbracket \circ \llbracket s_{k-1} \rrbracket \circ \cdots \circ \llbracket s_1 \rrbracket)(\varepsilon)$ is in the concretization of $X(\ell_k)$. Then, a refiner R_X is *sound* if σy is in the concretization of an abstract value in $R_X(\ell, y, \phi)$ for every concrete state $\sigma \in \llbracket \ell \rrbracket_{\langle \ell_0, T \rangle}$ that satisfies ϕ and is the final state of a trace that is abstracted by X and ends at ℓ.

[6]To improve performance, our implementation actually tracks these extra dependencies in a more fine-grained manner, as described in Section 6.3.

$$\textsc{Consequence}\quad \frac{\varphi_2' \Rightarrow \varphi_2 \qquad \langle\varphi_2'\rangle\ s\ \langle\varphi_1'\rangle \qquad \varphi_1 \Rightarrow \varphi_1'}{\langle\varphi_2\rangle\ s\ \langle\varphi_1\rangle}$$

$$\textsc{Frame}\quad \frac{\langle\hat{h}_1' \wedge \pi'\rangle\ s\ \langle\hat{h}_1 \wedge \pi\rangle \qquad \mathsf{mod}(s) \cap \mathsf{fv}(\hat{h}_2) = \emptyset}{\langle\hat{h}_1' * \hat{h}_2 \wedge \pi'\rangle\ s\ \langle\hat{h}_1 * \hat{h}_2 \wedge \pi\rangle}$$

$$\textsc{Disjunction}\quad \frac{\langle\varphi_l'\rangle\ s\ \langle\varphi_l\rangle \qquad \langle\varphi_r'\rangle\ s\ \langle\varphi_r\rangle}{\langle\varphi_l' \vee \varphi_r'\rangle\ s\ \langle\varphi_l \vee \varphi_r\rangle}$$

$$\textsc{BinOp}\quad \frac{\hat{h}\ =\ y \mapsto \hat{y} * z \mapsto \hat{z}}{\langle\hat{h} \wedge \pi \wedge \hat{x}\ =\ \hat{y} \oplus \hat{z}\rangle\ \ x = y \oplus z\ \ \langle\hat{h} * x \mapsto \hat{x} \wedge \pi\rangle}$$

$$\textsc{NewObj}\quad \frac{}{\langle\mathsf{unalloc}(\hat{x}) \wedge \pi \wedge (\textstyle\bigwedge_i \hat{z}_i = \mathsf{undef})\rangle\ \ x = \{\}\ \ \left\langle x \mapsto \hat{x} * \left(\mathop{\ast}_i \hat{x}[_] \mapsto \hat{z}_i\right) \wedge \pi\right\rangle}$$

$$\textsc{Alias}\quad \frac{}{\langle(y \mapsto \hat{y} \wedge \pi)[\hat{y}/\hat{x}]\rangle\ \ x = y\ \ \langle x \mapsto \hat{x} * y \mapsto \hat{y} \wedge \pi\rangle}$$

$$\textsc{ReadProp}\quad \frac{\hat{h}\ =\ y \mapsto \hat{y} * z \mapsto \hat{z} * \hat{y}[\hat{z}] \mapsto \hat{x}'}{\langle(\hat{h} \wedge \pi)[\hat{x}'/\hat{x}]\rangle\ \ x = y[z]\ \ \langle\hat{h} * x \mapsto \hat{x} \wedge \pi\rangle}$$

$$\textsc{Assume}\quad \frac{\hat{h}\ =\ x \mapsto \hat{x}}{\langle\hat{h} \wedge \hat{x} \wedge \pi\rangle\ \mathtt{assume}\ x\ \langle\hat{h} \wedge \pi\rangle}$$

$$\textsc{WriteProp}\quad \frac{\hat{h}\ =\ x \mapsto \hat{x} * y \mapsto \hat{y} * z \mapsto \hat{z}}{\langle(\hat{h} \wedge \pi)[\hat{z}/\hat{z}']\rangle\ x[y] = z\ \langle\hat{h} * \hat{x}[\hat{y}] \mapsto \hat{z}' \wedge \pi\rangle}$$

$$\textsc{Constant}\quad \frac{}{\langle\mathsf{true} \wedge \pi \wedge \hat{x} = p\rangle\ x = p\ \langle x \mapsto \hat{x} \wedge \pi\rangle}$$

Fig. 7. Rules for refutation-sound backwards abstract interpretation. We denote the set of memory locations possibly modified by a statement s by $\mathsf{mod}(s)$, the free variables of a heap constraint $\hat{h}$ by $\mathsf{fv}(\hat{h})$, and the substitution of symbolic variable $\hat{x}$ for $\hat{y}$ in a symbolic store φ by $\varphi[\hat{x}/\hat{y}]$. To simplify the presentation, without loss of generality, we assume the program has been normalized so that any statement involving multiple program variables uses distinct variables.

Soundness. Since we require that the refiner R_X is sound with respect to those traces abstracted by the base analysis state, $\mathcal{T}^{\mathsf{VR}}_{\ell \to_s \ell'}$ is sound when that base analysis state abstracts the full concrete semantics, which is guaranteed at a fixpoint. If the base analysis state does not yet abstract the full concrete collecting semantics, then R_X's refinements are only sound so long as the information they read from the base analysis state is unchanged. However, by adding additional dependency edges wherever such a read occurs, we ensure that any refinement query that used stale information will be invalidated and recomputed. We refer to Appendix A for a more detailed discussion.

5 BACKWARDS ABSTRACT INTERPRETATION FOR VALUE REFINEMENT

In this section, we define a sound value refiner $R^{\leftarrow}$ based on goal-directed backwards abstract interpretation. The value refiner $R^{\leftarrow}$ works by exploring backwards from the abstract state where it was triggered, overapproximating the set of states from which that state is reachable using an abstract domain based on separation logic constraints. We also detail the construction of a refiner $R^{\leftarrow}_X$ that extends $R^{\leftarrow}$ to access base analysis abstract state during value refinement.

This value refiner is *goal-directed* in the sense that it only traverses the subset of the control-flow graph relevant to a given refinement query and only computes transfer functions that directly affect its constraints. As such, it is quite precise with respect to a fixed property of interest without incurring the cost of applying that same precision to the full program.

5.1 Abstract Domain

The abstract domain of $R^{\leftarrow}$ is a constraint language over memory states. The syntax and semantics of this constraint language are given in Fig. 6. A *symbolic store* φ is a disjunctive normal form expression over heap constraints $\hat{h}$ and pure constraints π. Each clause represents a symbolic store, while the top-level disjunction permits case-splitting.

Heap constraints $\hat{h}$ are defined using an intuitionistic separation logic [Ishtiaq and O'Hearn 2001] wherein a single-cell heap constraint (i.e., $x \mapsto \hat{x}$ or $\hat{x}_1[\hat{x}_2] \mapsto \hat{x}_3$) holds for any heap containing that cell, not just for those heaps comprised only of that single cell. This results in a monotonic logic in which heap constraints $\hat{h}$ are preserved under heap extension. That is, if an intuitionistic separation logic assertion $\hat{h}$ holds for some concrete state σ, then $\hat{h}$ must also hold for all extensions σ' of σ. This succinctly supports a goal-directed analysis, in which we want to infer information only about some sub-portion of the heap.

Pure constraints π are either symbolic expressions $\hat{e}$ or conjunctions thereof; the pure constraint $\hat{e}$ holds whenever it could possibly evaluate to `true` according to the abstract expression evaluation function eval. By defining pure constraints over abstract values $\hat{v}$ rather than concrete values v, we will be able to seamlessly integrate information from the abstract state of the base analysis during refinement, as discussed in Section 4.3.

We denote by $\varphi \wedge \varphi'$ conjunction and re-normalization to DNF after alpha-renaming free symbolic variables in φ' such that all memory addresses are mapped to the same symbolic variable by both symbolic stores. For example, $(x \mapsto \hat{x} \wedge \hat{x} > 0) \wedge (x \mapsto \hat{y} \wedge \hat{y} < 5)$ reduces to $x \mapsto \hat{x} \wedge (\hat{x} > 0 \wedge \hat{x} < 5)$.

5.2 Backwards Abstract Interpretation

We define the analysis in terms of *refutation sound* [Blackshear et al. 2013] Hoare triples of the form $\langle\varphi\rangle\ s\ \langle\varphi'\rangle$, which are given in Fig. 7.

Refutation soundness is similar to the standard definition of soundness for Hoare logic (i.e., partial correctness), but in the opposite direction: a triple is refutation sound if and only if $((\sigma', \eta) \in \gamma(\varphi') \wedge [\![s]\!](\sigma) = \sigma') \Rightarrow (\sigma, \eta) \in \gamma(\varphi)$ holds for all σ, σ', and η. That is, a triple is refutation sound if any concrete run through s ending in a state satisfying the postcondition φ' must have started in a state satisfying the precondition φ.

These triples are best read from postcondition to precondition, since that is the natural direction in which to understand refutation soundness and the direction in which the analysis actually applies them.

The first three rules in Fig. 7 are integral structural components of the system that are not specific to any particular statement form. The Consequence rule allows the analysis to strengthen preconditions and weaken postconditions, making explicit our notion of refutation soundness and allowing other rules to materialize heap cells as needed; Frame enables local heap reasoning; and Disjunction splits reasoning over each disjunct of a symbolic store.

The remaining rules abstract the concrete semantics of their respective statement forms. ReadProp, WriteProp, and Alias transfer any postcondition constraints on their left-hand-side to precondition constraints on their right-hand-side; NewObj constrains any properties of the allocated object to be `undef` and asserts that the object is now unallocated, which ensures (by separation) that no such properties can be framed out by Frame; BinOp and Constant use pure equality constraints to precisely model the concrete semantics; and Assume directly encodes the assumption into a pure constraint.

Our value refiner $R^{\leftarrow} : Loc \times Var \times Constraint \rightarrow \mathcal{P}(\widehat{Val})$ is based on backwards abstract interpretation using the judgment $\langle\varphi\rangle\ s\ \langle\varphi'\rangle$. We introduce a distinguished symbolic variable res to represent the value that is being refined as we move backwards through the program. Note that

$R^{\leftarrow}$ is implicitly parameterized by a specific program $\langle \ell_0, T \rangle$, but does *not* have access to abstract states from the base analysis as discussed in Section 4.3; integration with the base analysis will be detailed in Section 5.3.

Given a refinement query $R^{\leftarrow}(\ell, x, y \mapsto \hat{v})$, we first encode the inputs as a symbolic store $x \mapsto \textsc{res} * y \mapsto \hat{y} \wedge \hat{y} = \hat{v}$. Then, we algorithmically apply the Hoare triples from Fig. 7 backwards from ℓ in T to compute a symbolic store for each backwards-reachable location, using a standard worklist algorithm to ensure correctness and minimize redundant work and applying the widening technique of Blackshear et al. [2013] to compute fixpoints over loops.

Due to refutation soundness, it is sound for the backwards abstract interpreter to stop at any point. Since each successive application of a rule from Fig. 7 computes an abstract precondition that a concrete execution must satisfy to reach the given abstract postcondition, the constraints on RES grow more precise the more of the program is analyzed but are overapproximate every step of the way. As such, the *stopping criterion* of $R^{\leftarrow}$ can be tuned, offering a tradeoff between refinement precision and performance. In our implementation for JavaScript, we stop the backwards traversal along a path if sufficient precision has been reached for the refinement variable RES, meaning that its abstract value is either a singleton set of object addresses or a non-$\top_{\text{Prim}}$ abstract primitive.

This analysis continues either until we reach a least fixpoint in the symbolic store domain (partially ordered under implication) or the stopping criterion is fulfilled for all symbolic stores in the worklist. At that point, we compute an upper bound on the value of RES in all remaining symbolic stores and return the corresponding set of abstract values.

Example. Recall that the base analysis issues three refinement queries for the example from Section 4.2, the first one being $R^{\leftarrow}(\ell_2, \mathtt{p}, \mathtt{t} \mapsto (\{\hat{a}_{\mathtt{xa}}\}, \bot_{\text{Prim}}))$. This query is encoded as the initial symbolic store $\mathtt{p} \mapsto \textsc{res} * \mathtt{t} \mapsto \hat{t} \;\wedge\; \hat{t} = (\{\hat{a}_{\mathtt{xa}}\}, \bot_{\text{Prim}})$ at the program location ℓ_2. From there, $R^{\leftarrow}$ uses the Consequence and ReadProp rules from Fig. 7 to construct the triple

$$\Big\langle\, \mathtt{p} \mapsto \textsc{res} * \mathtt{x} \mapsto \hat{x} * \hat{x}[\textsc{res}] \mapsto \hat{t} \;\wedge\; \hat{t} = (\{\hat{a}_{\mathtt{xa}}\}, \bot_{\text{Prim}}) \,\Big\rangle$$
$$\mathtt{t\ =\ x[p]}$$
$$\Big\langle\, \mathtt{p} \mapsto \textsc{res} * \mathtt{t} \mapsto \hat{t} \;\wedge\; \hat{t} = (\{\hat{a}_{\mathtt{xa}}\}, \bot_{\text{Prim}}) \,\Big\rangle$$

which precisely models the dynamic property read `t= x[p]` and yields a precondition symbolic store expressing a refinement of the value of `p` when `x[p]` has value $(\{\hat{a}_{\mathtt{xa}}\}, \bot_{\text{Prim}})$. We continue this example in the following section to show how the value refiner reaches the final result $\{(\emptyset, \texttt{"a"})\}$.

Soundness. By refutation soundness, applying the Hoare rules from Fig. 7 backwards soundly overapproximates the states from which the refinement location is reachable. By exhaustively tracking the value of RES on all backward abstract paths from the refinement location, $R^{\leftarrow}$ therefore computes an overapproximation of the variable being refined with respect to the concrete collecting semantics. We refer to Appendix B for a proof sketch.

5.3 Integration of Base Analysis State

We now extend $R^{\leftarrow}$ to leverage abstract state from the base analysis as described in Section 4.3, thereby constructing a value refiner $R^{\leftarrow}_X$ that is parameterized by a base analysis abstract state X. In particular, we describe a procedure by which a symbolic store φ and base analysis state X can be combined to compute a refinement that the symbolic store φ is not able to on its own. Essentially, when $R^{\leftarrow}_X$ has a symbolic store φ refining a property's name under a constraint on that property's value, it accesses the base analysis state to determine possible property names satisfying those constraints and then returns that set. We refer to this procedure as *property name inference*.

In more detail, this mechanism works as follows. If, during the backwards abstract interpretation as described for $R^{\leftarrow}$ in the previous section, the current symbolic store φ matches

$$x \mapsto \hat{x} * \hat{x}[\text{RES}] \mapsto \hat{y} * \hat{h} \ \wedge \ \hat{y} = \hat{v} \wedge \pi$$

for some x, $\hat{x}$, $\hat{y}$, $\hat{v} \neq \texttt{undef}$, $\hat{h}$, and π, then property name inference is applied. Intuitively, this condition means that the abstract value of the RES property of x is $\hat{v}$, which allows $R_X^{\leftarrow}$ to determine the desired set of property names by reading the base analysis state at that location. We define a function infer-prop-names to compute that refinement:

$$\text{infer-prop-names}(\varphi, \hat{\sigma}) = \left\{ p \;\middle|\; \begin{array}{c} \exists \hat{a} : \hat{a} \prec_1 \hat{\sigma} x \ \wedge \ (\hat{a}, p) \in \text{dom}(\hat{\sigma}) \\ \wedge \ \hat{\sigma}(\hat{a}, p) \sqcap \hat{v} \neq (\emptyset, \bot_{Prim}) \end{array} \right\}$$

Intuitively, infer-prop-names checks each property $(\hat{a}, p)$ on the object x, returning the names p of those properties whose abstract value intersects with $\hat{v}$.

The property name inference mechanism thus refines the abstract *names* of object properties. Our implementation uses the same idea to also refine abstract *values* of properties, which we return to in Section 6.2.

Example. Continuing the example from Section 5.2, φ is of the form specified above, so the analysis applies property name inference to compute a refinement for `p`. Computing the refinement infer-prop-names$(\varphi, X(\ell_1))$ gives the names of those properties in $X(\ell_1)$ that satisfy φ, meaning that the property value intersects with $(\{\hat{a}_{\texttt{xa}}\}, \bot_{\text{Prim}})$. In this case, `"a"` is the only such property name according to the value of $X(\ell_1)$ given in Section 3.2, so the refiner returns $\{(\emptyset, \texttt{"a"})\}$. In this simple example, a single step backwards suffices before the stopping criterion is fulfilled, due to the integration of the base analysis state, but multiple steps are often needed in practice.

6 INSTANTIATION FOR JAVASCRIPT

Our implementation, TAJS_{VR},[7] generalizes to JavaScript the ideas presented in the previous sections for the simple dynamic language. As base dataflow analysis TAJS_{VR} uses the existing tool TAJS, extended as explained in Section 4.3. The other main component of the implementation is the value refiner, built from scratch and based on the design given in Section 5. The two components are implemented separately – the base analysis in Java (approximately 2500 lines of code on top of TAJS) and the refiner in Scala (approximately 2400 lines of code) – and communicate only through a minimal interface that allows the base analysis to issue refinement queries and the refiner to read partially-computed base analysis state, request control-flow information to traverse the program, or perform property name inference as described in Section 5.3. The implementation and experimental data are available at https://www.brics.dk/TAJS/VR.

6.1 A Value Refiner for JavaScript

Many JavaScript language features that are not directly in the minimal dynamic language are straightforward to handle. However, `for-in` loops, interprocedural control flow, and prototype-based inheritance are nontrivial and require some additional machinery in the backwards analysis.

`for-in` *loops.* In order to handle `for-in` loops efficiently, the refiner analyzes the loop body under contexts corresponding to the properties of the loop object in the base analysis state.

That is, upon reaching the exit of a `for-in` loop, the value refiner queries the base analysis state for a set of property names on the loop object, generating one context for each of them and one additional context as a catch-all for all other property names. Then, it analyzes the loop body once per context before joining the results and continuing backwards from the loop entry.

[7] **TAJS** with demand-driven **V**alue **R**efinement

Interprocedural control flow. In order to soundly navigate interprocedural control flow in the value refinement analysis, we rely on the partially-computed call graph from the base analysis while maintaining a stack of return targets where possible. That is, when reaching a call site, the analysis pushes that location (along with any locally-scoped constraints) onto a stack before jumping to the exit of all possible callees in the base analysis call graph. Then, upon reaching a function entry point, it pops a stack element and jumps to the corresponding call site or, if the stack is empty, jumps to all callers of the current function in the base analysis call graph. When using an unbounded stack, it analyzes function calls fully context-sensitively and therefore relies on a timeout to ensure termination, but k-limiting the stack height would ensure termination (without a timeout) while analyzing function calls with k-callstring context sensitivity.

Prototype-based inheritance. Handling prototype-based inheritance is more complicated since the semantics of a dynamic property access depend not only on the values of the object and property name but also on the prototype relations and properties of other objects in the program.

Our implementation reasons about prototype-based inheritance by introducing "prototype constraints" at property reads to keep track of prototype relationships between relevant symbolic variables. These constraints are manipulated as the analysis evaluates other property writes and modifications to the prototype graph; for example, when encountering a property write under a prototype constraint, the analysis splits into a disjunction on whether or not the write is to the memory location whose read produced the prototype constraint. This allows the analysis to reason about prototype semantics, even in programs that dynamically modify the prototype graph.

6.2 Functions with Free Variables

As mentioned in Section 2, the dynamic property read/write pair ② in the example in Fig. 1a differs from ① and ③, because the values flow from the property read to the associated write via a free variable that is declared in an enclosing function. It is critical that the analysis does not mix together the different functions of the `source` object. For example, in clients of Lodash, the function value `lodash([1, 2]).map` is the one created in line 7 where `methodName` is `"map"`. If the program contains a call to that function, then at the call `func.apply(...)` in line 9, the analysis must have enough precision to know that `func` is the same function as `source["map"]`.

We achieve that degree of precision by adjusting the base analysis as follows. At the dynamic property write in line 7, the analysis detects that in the current abstract state, the property name (i.e. `methodName`) is imprecise and that the value being written denotes a function that contains a free variable with an imprecise value as noted in Section 2. The analysis then annotates the abstract value being written with the memory address of `methodName` and the current program location ℓ_7 for later use. Every property of `object.prototype` then has this single annotated abstract value.

When the analysis later encounters a property read operation that yields such an annotated value, the value is modified to reflect the property name, which can now be resolved. For example, at an expression `lodash([1, 2]).map`, the resulting abstract value describes a function that has been created at a point where the value of `methodName` was `"map"`.

If that function is called, the analysis reaches line 9 and then issues the refinement query $R_X^{\leftarrow}(\ell_7, \texttt{func}, \texttt{methodName} \mapsto \texttt{"map"})$ to learn the possible values of `func` at line 7 under the constraint that `methodName` has the value `"map"`. When the value refiner reaches the dynamic property read in line 4 during the backwards analysis, it can use the base analysis state to read `source["map"]` (as hinted in Section 5.3), which provides the desired precise value for `func`.

Note that with this mechanism, our base analysis does not only trigger refinement at dynamic property writes as described in Section 4.2, but also for variable reads of imprecise free variables as described above.

6.3 Performance Improvements

Recall from Section 4.3 that a location ℓ is added to the worklist when a refinement query at ℓ has accessed the abstract state $X(\ell')$ and that abstract state has changed. Our implementation uses a more fine-grained notion of dependencies by keeping track of the individual dataflow facts instead of entire abstract states, such that ℓ is only added to the worklist when the state change at ℓ' invalidates the dataflow facts that were previously accessed from ℓ'.

Additionally, our implementation caches refinement query results, exploiting the fact that the result of a query $R_X(\ell, y, \phi)$ depends only on the three parameters and the dataflow facts in X that are accessed by the value refiner.

7 EVALUATION

We evaluate the demand-driven value refinement technique by considering the following research question:

> Can TAJS$_{\text{VR}}$ analyze programs that other state-of-the-art tools are unable to analyze soundly and with high precision?

To provide insights into why the mechanism is effective when analyzing real-world programs, we also investigate how many value refinement queries are issued, how often the value refiner is able to produce more precise results than the base analysis, and how much analysis time is typically spent on value refinement.

7.1 Comparison with State-of-the-Art Analyzers

We compare TAJS$_{\text{VR}}$ with two existing state-of-the-art analysis tools: TAJS [Andreasen and Møller 2014; Jensen et al. 2009] and CompAbs [Ko et al. 2017, 2019]. TAJS is the base dataflow analysis upon which TAJS$_{\text{VR}}$ is built; it is designed for JavaScript type analysis but performs no value refinement. CompAbs – described in further detail in Sections 2 and 8 – is a tool built on top of SAFE [Lee et al. 2012] that attempts to syntactically identify problematic dynamic property access patterns and applies trace partitioning at those locations.

We evaluate each tool on three sets of benchmarks: a series of micro-benchmarks designed as minimal representative examples of dynamic property manipulation patterns, a collection of evaluation suites drawn from other JavaScript static analysis research papers, and the unit test suites of two popular JavaScript libraries that are unanalyzable by the existing static analysis tools. All experiments have been performed on an Ubuntu machine with 2.6 GHz Intel Xeon E5-2697A CPU running a JVM with 10 GB RAM. Collectively, the results indicate that the relational information provided by value refinement is critical for the analysis of challenging JavaScript programs.

Micro-Benchmarks. Following the approach of Ko et al. [2017], we first evaluate TAJS$_{\text{VR}}$ on a series of small benchmarks containing dynamic property access patterns known to be difficult for static analysis. Source code for these benchmarks can be found in Ko et al. [2017] (CF, CG, AF, and AG) or at https://www.brics.dk/TAJS/VR (M1, M2, and M3).

The results, shown in Table 1, are as expected: TAJS only handles the benchmarks CF and CG where the property names are known statically, and CompAbs fails to successfully analyze any of M1, M2 or M3: M1 and M3 because the relevant read/write pair is not detected by the syntactic patterns, and M2 because the trace partitioning mechanism does not distinguish closures on free variables within the partitions. TAJS$_{\text{VR}}$ handles all seven programs precisely by the use of demand-driven value refinement.

Table 1. Micro-benchmarks that check how state-of-the-art analyses handle various dynamic property access patterns. A ✗ indicates that the analysis mixes together the properties of the object being manipulated, while a ✓ indicates that it is sufficiently precise to keep them distinct. The CF, CG, AF, and AG benchmarks are drawn directly from [Ko et al. 2017], while M1, M2, and M3 are isolated and distilled from the read/write patterns presented in Section 2.

Benchmark		TAJS	CompAbs	TAJS_{VR}
CF	(`for-in` loop over statically known set of properties)	✓	✓	✓
CG	(`while` loop over statically known set of properties)	✓	✓	✓
AF	(`for-in` loop over statically unknown properties)	✗	✓	✓
AG	(`while` loop over statically unknown properties)	✗	✓	✓
M1	(indirect field copy, distilled from ① in Fig. 1)	✗	✗	✓
M2	(field copy through closure, distilled from ② in Fig. 1)	✗	✗	✓
M3	(interprocedural field copy, distilled from ③ in Fig. 1)	✗	✗	✓

Library Benchmarks. Prior work has found that the analysis of highly dynamic, metaprogramming-heavy libraries is a major hurdle for the analysis of realistic JavaScript programs in the wild [Andreasen and Møller 2014; Park and Ryu 2015; Sridharan et al. 2012]. As such, we evaluate TAJS_{VR} against TAJS and CompAbs on a corpus of challenging real-world library benchmarks, drawn from the benchmark suites of past JavaScript analysis research works as well as from library unit test suites. We selected these benchmarks to evaluate our approach on programs that other researchers find important and to demonstrate that our approach enables the analysis of programs beyond the reach of existing analyzers.

From past works, we analyze the jQuery tests found in Andreasen and Møller [2014], the Prototype.js and Scriptaculous tests from Wei et al. [2016], and the test suite (excluding 7 benchmarks that require additional modelling of Firefox add-ons) used by Kashyap et al. [2014] and Dewey et al. [2015] (referred to collectively as "JSAI tests" henceforth).

In addition, we analyze the unit test suites for two heavily used functional utility libraries: Underscore (v1.8.3, 1548 LoC) Lodash3 (v3.0.0, 10785 LoC), and Lodash4 (v4.17.10, 17105 LoC). The unit tests (664 in total) provide comprehensive coverage and illustrate realistic use-cases of the two libraries. We select Lodash and Underscore because they are the two most-depended-upon packages in NPM[8] that do not require platform-specific modelling for Node.js and are unanalyzable by TAJS without the use of value refinement. In total, more than 100,000 NPM packages depend on one or both libraries (excluding transitive dependencies), so they represent a significant hurdle for analysis of Node.js modules and applications. Both Lodash3 and Lodash4 are included since their codebases are substantially different and they present distinct challenges for static analysis.

A summary of the results of each tool on these library benchmark is given in Table 2. We say that a program is *analyzable* when analysis terminates within 5 minutes, and with dataflow to the program exit. In our experiments, we find that increasing this time budget does not allow the tools to successfully analyze many more tests, due to the all-or-nothing nature of dynamic language analysis: analyzers are generally either precise enough to analyze a program quickly, or they lose precision at some key location, leading to a proliferation of spurious dataflow that renders the analysis results useless and cannot be recovered from regardless of time budget. This phenomenon has also been observed by Jensen et al. [2009], Park and Ryu [2015], and Ko et al. [2017]. For some of the tests, the CompAbs tool does terminate quickly but has no dataflow to the program exit, which is unsound for these programs. In comparison, TAJS_{VR} passes extensive soundness testing as explained below.

[8] At time of publication, Lodash ranks #1 and Underscore #14, per https://npmjs.com/browse/depended.

Table 2. Analysis results for real-world benchmarks drawn from previous evaluations of JavaScript analysis tools [Andreasen and Møller 2014; Kashyap et al. 2014; Wei et al. 2016] and additional library unit test suites. A test is a "Success" if the analysis terminates with dataflow to the program exit within a 5 minute timeout, and times are averaged across all successfully analyzed tests.

Benchmark		**TAJS**		**CompAbs**		**TAJS_{VR}**	
		Success (%)	Time (s)	Success (%)	Time (s)	Success (%)	Time (s)
JQuery	(71 tests)	7%	14.4	0%	-	7%	17.2
JSAI tests	(29 tests)	86%	12.3	34%	32.4	86%	14.3
Prototype	(6 tests)	0%	-	33%	23.1	83%	97.7
Scriptaculous	(1 test)	0%	-	100%	62.0	100%	236.9
Underscore	(182 tests)	0%	-	0%	-	95%	2.9
Lodash3	(176 tests)	0%	-	0%	-	98%	5.5
Lodash4	(306 tests)	0%	-	0%	-	87%	24.7

Demand-driven value refinement enables TAJS_{VR} to efficiently analyze many benchmarks that TAJS cannot handle. The Prototype and Scriptaculous libraries are unanalyzable by TAJS, but the relational information provided by value refinement allows TAJS_{VR} to successfully analyze the Scriptaculous test and five of six Prototype tests. For the tests that CompAbs can analyze, it is faster than TAJS_{VR}. Less than 5% of the analysis time for TAJS_{VR} is spent performing value refinement for those benchmarks, so refinement is not the primary reason for this difference; we believe that the reason is rather that CompAbs uses a more precise model of the DOM, which is used heavily in both libraries.

TAJS_{VR} is furthermore able to analyze 92% (611/664) of the Underscore, Lodash3, and Lodash4 unit tests, none of which are analyzable by either TAJS or CompAbs, in 13 seconds on average. This result is explained in part by the analysis behavior on the micro-benchmarks above. Since the M1, M2, and M3 micro-benchmarks are extracted from Lodash library bootstrapping code and neither TAJS nor CompAbs can reason precisely about them, it follows that neither tool can precisely analyze the library test cases. The result also indicates that the relational information provided by value refinement – and, correspondingly, TAJS_{VR}'s ability to analyze the M1, M2, and M3 micro-benchmarks – is integral to the precise analysis of libraries like Underscore and Lodash.

Manual triage shows that the library unit tests that TAJS_{VR} fails to analyze are mostly due to challenges orthogonal to dynamic property access operations. Some of the tests involve complex string manipulations, some are caused by insufficient context sensitivity in the base dataflow analysis, and most of the remaining ones could be handled by improving TAJS' reasoning at type tests in branches. Also, our value refiner fails to provide sufficiently precise answers for approximately 0.02% of queries, as discussed in Section 7.2.

For those benchmarks that TAJS can handle without demand-driven value refinement, TAJS_{VR} provides similar results to TAJS, both in terms of precision and performance. Because the static determinacy technique by Andreasen and Møller [2014] enables TAJS to analyze many of jQuery's dynamic property writes precisely, the jQuery test cases where TAJS_{VR} fails are unanalyzable for reasons unrelated to dynamic property accesses and so value refinement is rarely triggered. The results for the JSAI tests are analogous: since TAJS can reason precisely about them without value refinement for the most part, TAJS_{VR} yields similar results to TAJS and never triggers refinement. More data about the refinement queries issued for these benchmarks are presented in Section 7.2. Overall, the results indicate that extending an analysis with demand-driven value refinement does not add significant cost in situations where the base analysis is already sufficiently precise.

Precision. We measure TAJS_{VR} analysis precision with respect to type analysis and call graph construction, following the methodology of prior JavaScript analysis works [Andreasen and Møller 2014; Park and Ryu 2015] that have established these metrics as useful proxies for precision. In these measurements, we treat locations context-sensitively, counting the same location once per context under which it is reachable. At each variable or property read in a program successfully analyzed by TAJS_{VR}, we count the number of possible types for the resulting value; in 99.48% of cases, the value has a single unique type, and the average number of types per read is 1.009 (of course, the actual value must be at least 1 at every read). Similarly, we measure the number of callees per callsite, finding that 99.95% of calls have a unique callee.

For analysis of a library to be useful it is also important that the library object itself is analyzed precisely, such that properties of the library yield precise values when referenced in client programs. To verify that is the case, we check all methods of library objects (i.e., properties that contain functions) in programs successfully analyzed by TAJS_{VR}. We find that 99.44% of such methods contain a unique function, indicating that TAJS_{VR} successfully avoids mixing together the library methods.

These numbers clearly demonstrate that in the situations where the critical precision losses are avoided and the analysis terminates successfully, the analysis results are very precise. This degree of precision may enable analysis clients such as program optimizers and verification tools; however, developing such client tools is out of scope of this work.

Soundness Testing. To increase confidence that TAJS_{VR} is sound, we have applied the empirical soundness testing technique of Andreasen et al. [2017]. The technique checks whether the analysis result overapproximates all values observed in every step of a concrete execution. For example, if the program at some point in the execution writes the number 42 to a property of an object, then the analysis must at that point have an abstract value that overapproximates that concrete value. Since most of the benchmarks do not require user interaction, a single concrete execution for each suffices to get good coverage. In total, more than 7.8 million pairs of concrete and abstract values are tested. Only 117 of them fail, all for the same reason: one Lodash4 test uses `Arrays.from` in combination with ES6 iterators, which is not fully modeled in the latest version of TAJS.

Scalability Compared to Trace Partitioning. When analyzing the initialization code of Lodash4, TAJS_{VR} only issues value refinement queries at the dynamic property writes in ① and ③ in Fig. 1. The precise information provided by these queries can also be gained using the trace partitioning technique from CompAbs at the correlated reads of ① and ③. To compare the scalability of value refinement with that of trace partitioning (isolated from the choice of where to issue value refinement queries or apply trace partitioning), we have implemented an extension of TAJS that is hardcoded to perform trace partitioning at exactly those two reads.

TAJS_{VR} analyzes the initialization code of Lodash4 in 19 seconds while TAJS with hardcoded trace partitioning takes 222 seconds. This result indicates that value refinement scales better than trace partitioning even when partitioning is applied only at the necessary locations.

7.2 Understanding the Effectiveness of the Value Refiner

Table 3 shows statistics about the value refinement queries that are issued when analyzing the benchmark suites.[9] In summary, these results demonstrate that value refinement queries are being triggered at only a small fraction of all the dynamic property write operations, and that the backwards abstract interpreter described in Sections 4 and 6.1 is an effective value refiner: it efficiently computes highly precise refinements on the base analysis state in the vast majority of cases, often spending only a few milliseconds and visiting only a small part of the program.

[9] More granular experimental results can be found in Appendix C.

Table 3. Summary of value refinement behavior for library tests. "Ref. locs" is the total number of program locations where refinement queries are issued out of the total number of property writes with dynamically-computed property names; "Avg. queries" is the average number of queries issued per test; "Success" is the percentage of queries where the value refiner produces a result more precise than the base analysis state for the requested memory address; "Refiner time" is the percentage of the total analysis time spent by the value refiner; "Avg. query time" is the average time spent by the value refiner on each query; "Avg. locs visited" is the average number of program locations visited in each invocation of the value refiner; "Inter." is the percentage of the queries where the value refiner visits multiple functions; and "PNI" is the percentage of queries where the value refiner uses property name inference (Section 5.3).

		Ref. locs	Avg. queries	Success (%)	Refiner time (%)	Avg. query time (ms)	Avg. locs visited	Inter. (%)	PNI (%)
JQuery	(71 tests)	5 / 138	1.13	87.5	0.1	13.57	7.1	2.86	90.00
JSAI tests	(29 tests)	0 / 2705	-	-	-	-	-	-	-
Prototype	(6 tests)	4 / 69	188.17	100.0	2.5	13.08	39.98	48.10	97.61
Scriptaculous	(1 test)	2 / 92	601.00	100.0	3.4	13.21	36.91	42.26	99.33
Underscore	(182 tests)	5 / 32	267.84	99.98	22.4	2.43	5.05	0.10	99.76
Lodash3	(176 tests)	12 / 132	475.28	99.99	47.2	5.46	10.47	40.22	99.90
Lodash4	(306 tests)	7 / 123	1284.04	99.97	52.0	10.01	10.09	25.75	99.67

Semantic Triggers for Refinement. Value refinement is triggered (meaning that the first case in the definition of the modified transfer function $\mathcal{T}^{\mathsf{VR}}_{\ell \to_s \ell'}$ applies) at a total of only 35 program locations across the 7 benchmark groups. This is a low number compared to the sizes of the benchmarks (which contain a total 3291 property writes with dynamically computed property names), but as we have seen in Section 7.1, adequate relational precision at those 35 locations is critical for successful analysis of library clients.

The value refiner is invoked many times for the benchmarks that TAJS cannot analyze without refinement but quite rarely in the benchmarks (JSAI and JQuery) that TAJS can handle alone. As discussed in Section 7.1, this is because we trigger refinement semantically only at imprecise dynamic property writes, but TAJS has sufficient precision to avoid imprecise writes without applying refinement in some benchmarks. As for the large number of refinement queries for the other benchmarks, recall that each computation of $\mathcal{T}^{\mathsf{VR}}_{\ell \to_s \ell'}$ issues multiple refinement queries for the same memory locations under different constraints.

Effectiveness of Backwards Abstract Interpretation. The table shows that over 99% of refinement queries are successful, in the sense that the value refiner computes a more precise abstract value for the queried memory location than the base analysis alone. Each invocation of the value refiner is limited to 2 seconds, and all unsuccessful queries reported in Table 3 are due to this timeout. Even though we invoke the value refiner often, in most cases less than half of the total analysis time is spent performing value refinement, and that fraction of time only exceeds 4% for the Underscore and Lodash experiments. This low performance cost of refinement is due to the fact that most queries are answered in a few milliseconds and only require the backwards analysis to visit quite few program locations: the average for every one of the seven benchmark groups is below 40 locations, even in programs containing thousands of lines of code. This indicates that the "goal-directed" nature of the backwards abstract interpreter – the fact that it reasons only about the portion of the heap relevant to the query at hand and traverses only the subset of the control-flow graph needed to answer that query – is integral to its effectiveness. Still, many queries require interprocedural reasoning, especially for Lodash, Prototype, and Scriptaculous.

We additionally observe that property name inference (the mechanism described in Section 5.3, in which we leverage the abstract states of the base analysis when performing value refinement) is used by almost all queries. That mechanism is not only used often by TAJS_{VR}, it is essential to its success; an extra experiment shows that if we disable property name inference, then TAJS_{VR} fails on all of the same tests as TAJS in Table 2.

A manual investigation has shown that in situations where the value refiner fails to provide precise results, the relevant code does not require relational information between the property name in the write and the value to be written. A typical example is the code snippet `result[pad + h[hIndex]] = args[argsIndex++]` from Lodash3. Since there is no relation between `hIndex` and `argsIndex`, the value refinement mechanism is unable to precisely resolve the dynamic property write operation and the value being written. Conversely, when the refinement does hinge on relational information, the value refiner is precise and effective in solving the queries.

8 RELATED WORK

There is a wealth of research on techniques for static analysis of JavaScript applications, much of which has focused on the development of general analysis frameworks such as TAJS [Jensen et al. 2009], SAFE [Lee et al. 2012], WALA [IBM Research 2018], and JSAI [Kashyap et al. 2014].

Notably, much recent work has focused on improving analysis precision with various algorithmic innovations. The static determinacy technique by Andreasen and Møller [2014] and the loop-sensitive analysis by Park and Ryu [2015] both employ specialized context-sensitivity policies to achieve greater precision, especially in loop bodies and for free variables in closures. The correlation tracking points-to analysis by Sridharan et al. [2012] and the composite abstraction by Ko et al. [2017, 2019] target the dynamic behaviors exhibited by particularly troublesome syntactic patterns and apply extra precision wherever those syntactic patterns are detected.

All of the JavaScript analysis tools mentioned above are whole-program analyzers, while successful analyses for other languages typically achieve scalability by modularity. For dynamic programming languages like JavaScript, it is hard to achieve modularity without involving complex specifications of the program components. Moreover, even a modular analysis inevitably has to reason about the extremely difficult pieces of code that exist in libraries like those studied in this paper.

Other researchers have developed abstractions to better model JavaScript's idiosyncratic heap semantics. Cox et al. [2014] have introduced a complex relational abstraction to reason more precisely and efficiently about open objects, and Gardner et al. [2012] and Santos et al. [2018, 2019] have designed a separation logic-based program logic and verification engine to succinctly express and verify heap properties. These approaches are modular but require complex, manually provided specifications, whereas TAJS_{VR} is fully automatic.

In the area of demand-driven and refinement analysis, much of the prior work has targeted pointer analyses for languages like Java [Guyer and Lin 2005; Liang and Naik 2011; Späth et al. 2016; Sridharan and Bodík 2006]. These works each employ some form of abstraction refinement to increase analysis precision as needed, successively tightening their overapproximations of the forward concrete semantics. Similarly, the technique by Gulavani and Rajamani [2006] symbolically propagates error conditions backwards to generate hints for choosing between joining or widening in a finite powerset domain. Like standard counterexample-guided abstraction-refinement [Ball and Rajamani 2001; Clarke et al. 2000; Henzinger et al. 2002], these techniques alternate between deriving counterexamples and restarting the fixpoint analysis with a refined abstraction. In contrast, our approach interleaves refinement queries within a single fixpoint analysis and refines the abstract values instead of the analysis abstraction.

Several recent techniques use refutation sound backwards analyses for refinement. For example, Blackshear et al. [2013] use a backwards abstract interpreter to refute false alarms from a base pointer analysis, and Cousot et al. [2011] infer preconditions by backwards symbolic propagation of program assertions. These works are distinct from the well-known backward techniques for weakest-precondition analysis [Chandra et al. 2009; Flanagan et al. 2002; Manevich et al. 2004], which are underapproximate and use a forward soundness condition that is dual to refutation soundness (as described in Section 5.2). See also Ball et al. [2005] for a more detailed treatment of these distinct soundness properties; in their terminology, the forwards soundness condition corresponds to a *must*$^{+}$ transition while refutation soundness corresponds to a *must*$^{-}$ transition.

Our work is most similar to that of Blackshear et al. [2013], but generalizes their technique in several ways. Whereas their tool is used after-the-fact for alarm triage only, our demand-driven analysis runs during the execution of the base analysis; we generalize from boolean refutation queries of the form "is this abstract state reachable?" to value refinement queries of the form "which values can this memory location hold at this abstract state?"; and we extend their technique from Java to JavaScript, where dynamic property access introduces significant new analysis challenges for the value refiner.

Combining separate static analyses has also been the subject of much research, going back to the introduction of the reduced product abstract domain by Cousot and Cousot [1979]. Much of the recent work on combinations of analyses has similarly focused on combining abstract domains while leaving the analysis algorithm itself largely unchanged, essentially composing analyses in parallel [Chang and Leino 2005; Cousot et al. 2006; Lerner et al. 2002; Toubhans et al. 2013]. Other works have composed analyses in series, for example to guide context-sensitivity policies using a pre-analysis [Oh et al. 2016] or to filter spurious alarms from a coarse whole-program analysis with a targeted refutation analysis [Blackshear et al. 2013]. Our framework, in which a forward analysis communicates and interleaves with a backward analysis that is triggered on demand when high precision is needed, cannot easily be expressed within these existing formalisms for combining abstract domains or sequencing analyses.

9 CONCLUSION

We have presented a novel program analysis mechanism, *demand-driven value refinement*, and shown how it can be used to soundly regain lost precision during the execution of a dataflow analysis for JavaScript programs. The mechanism is particularly effective at providing precise relational information, even when the base dataflow analysis employs a non-relational abstract domain.

In experiments using our implementation TAJS_{VR}, we have demonstrated that demand-driven value refinement is more effective than a state-of-the-art alternative technique that relies on syntactic patterns and trace partitioning, when analyzing widely used JavaScript libraries. These results suggest that demand-driven value refinement is a promising step towards fast and precise static analysis for real-world JavaScript programs.

A SOUNDNESS OF DEMAND-DRIVEN VALUE REFINEMENT

First, we argue that the base analysis remains sound when switching to the modified transfer functions $\mathcal{T}^{\mathsf{VR}}_{\ell \to_s \ell'}$ using R, provided that R is sound. Second, we extend this result from R to R_X.

First, note that if $R(\ell, y, z \mapsto \hat{z})$ always returns $\hat{\sigma}(y)$, then the definition of $\mathcal{T}^{\mathsf{VR}}_{\ell \to_s \ell'}$ yields a node transfer function that is equivalent to the original one $\mathcal{T}_{\ell \to_s \ell'}$. Therefore, $\mathcal{T}^{\mathsf{VR}}_{\ell \to_s \ell'}$ is sound provided that it is sound to use an actual refiner, instead of one always returning $\hat{\sigma}(y)$.

Since R is sound by assumption, we have – per the definition of refiner soundness in Section 4.1 – that for every state $\sigma \in [\![\ell]\!]_{\langle \ell_0, T\rangle}$ where ℓ is a location in the program $\langle \ell_0, T\rangle$ and the abstraction of σ satisfies the constraint $z \mapsto \hat{z}$, the value σy is in the concretization of an abstract value in $R(\ell, y, z \mapsto \hat{z})$ for any y. That is, the refinement result over-approximates the possible concrete values of y at ℓ under the given constraint. As such, so long as the partitioning $Part(\hat{\sigma} z)$ overapproximates z, $V(\hat{\sigma}, \ell, y, z, p)$ overapproximates the possible values written to $\hat{m}$ and the base analysis therefore remains sound.

On the other hand, if the base analysis receives new dataflow that invalidates the partitioning (i.e., such that $\hat{\sigma} z$ is no longer covered by the partitioning) at ℓ, then $V(\hat{\sigma}, \ell, y, z, p)$ may not be overapproximate. However the dataflow update at ℓ causes the refinement to rerun with a new partitioning that does cover the new $\hat{\sigma} z$.

All such new dataflow will eventually be processed, so eventually the partitioning $Part(\hat{\sigma} z)$ will overapproximate z at ℓ. Thus, from trace partitioning, the refinement result is sound with respect to the collecting semantics and $V(\hat{\sigma}, \ell, y, z, p)$ therefore bounds the possible value written to $\hat{m}$.

We will now extend this to show that it is sound to use R_X instead of R, meaning that the value refiner can use dataflow facts from the base analysis.

Note that, by the definition of R_X soundness in Section 4.3, R_X overapproximates the full collecting semantics when the base analysis has abstracted all traces. As such, soundness with R_X follows from soundness with R once the base analysis has abstracted all traces. Therefore, we only need to argue the case where the base analysis issues a refinement query at ℓ_r, R_X accesses the base analysis' state at ℓ_d, and there exists a concrete trace $\ell_0 \rightarrow_{s_1} \cdots \rightarrow_{s_d} \ell_d \rightarrow \cdots \rightarrow_{s_r} \ell_r$.

If all partial concrete traces to ℓ_d are abstracted by the base analysis state (as defined in Section 4.3), then the abstract state read at ℓ_d overapproximates the possible concrete states at ℓ_d and the refinement result is sound. Otherwise, there exists a trace $\ell_0 \rightarrow_{s_1} \cdots \rightarrow_{s_d} \ell_d$, that is not yet abstracted by the base analysis. When this trace gets abstracted by the base analysis, we will have new dataflow at ℓ_d. At that point, ℓ_r will be added to the worklist due to the updated worklist dependency map of Section 4.3. That is, ℓ_r will always be re-added to the worklist if a dataflow fact it depended on was from a not-yet-overapproximating base analysis state.

Thus, in the final transfer function execution at ℓ_r, all the dataflow facts used by the value refiner will be overapproximate with respect to the collecting semantics. Therefore, the refinement result is overapproximate as well by refiner soundness, so $V(\hat{\sigma}, \ell, y, z, p)$ bounds the possible concrete values written to $\hat{m}$ and the base analysis therefore remains sound when using the modified transfer functions $\mathcal{T}^{\mathsf{VR}}_{\ell \rightarrow_s \ell'}$ and the refiner R_X.

B SOUNDNESS OF THE VALUE REFINER $R^{\leftarrow}$

Let us write $v \models \hat{v}$ for v being in the concretization of $\hat{v}$. Then, following the definition of refiner soundness given in Section 4, it suffices to show, for all variables x, program locations ℓ, and constraints $y \mapsto \hat{v}$, that if a concrete state σ such that $\sigma y \models \hat{v}$ is in the collecting semantics $[\![\ell]\!]_{\langle \ell_0, T\rangle}$, then there exists a corresponding $\hat{u}$ in the refinement $R^{\leftarrow}(\ell, x, y \mapsto \hat{v})$ such that $\sigma x \models \hat{u}$.

Take any $\ell, x, y \mapsto \hat{v}$, and σ, supposing that $\sigma y \models \hat{v}$ and $\sigma \in [\![\ell]\!]_{\langle \ell_0, T\rangle}$. Since σ is in the concrete semantics at ℓ, there exists a trace leading to that state: $\ell_0 \rightarrow_{s_0} \ell_1$, ... , $\ell_n \rightarrow_{s_n} \ell \in T^*$ where $([\![s_n]\!] \circ [\![s_{n-1}]\!] \circ \cdots \circ [\![s_0]\!])(\varepsilon) = \sigma$.

Let φ be the initial abstract store given in the definition of $R^{\leftarrow}$. That is, let $\varphi = x \mapsto \textsc{res} * y \mapsto \hat{y} \wedge \hat{y} = \hat{v}$. Note that, by construction, there exists $(\sigma, \eta) \in \gamma\varphi$ such that $\eta(\textsc{res}) = \sigma x$.

Let φ_i be the symbolic store computed at ℓ_i by the fixpoint algorithm. By refutation soundness, it must be the case that $\langle \varphi_n \rangle s_n \langle \varphi \rangle$, $\langle \varphi_{n-1} \rangle s_{n-1} \langle \varphi_n \rangle$, and so on through $\langle \varphi_0 \rangle s_0 \langle \varphi_1 \rangle$.

Transitively applying the refutation soundness property for each of these triples yields the implication $(\sigma, \eta) \in \gamma\varphi \Rightarrow (\varepsilon, \eta) \in \gamma\varphi_0$. Therefore, there exists $(\varepsilon, \eta) \in \gamma\varphi_0$ with $\eta(\textsc{res}) = \sigma x$.

Therefore, the upper bound $\bigsqcup_{(\sigma,\eta)\in\gamma(\varphi_0)} \beta(\eta(\textsc{res}))$ (where $\beta : \mathrm{Val} \to \widehat{\mathrm{Val}}$ denotes value abstraction) on the value of RES that is computed for the symbolic store φ_0 overapproximates σx and thus the value refinement $R^{\leftarrow}(\ell, x, y \mapsto \hat{u})$ soundly overapproximates σx.

C SUPPLEMENTARY EXPERIMENTAL DATA

Table 3 presented accumulated statistics for all of the refinement queries performed during the analysis of the library benchmarks. In this section, we provide more granular details about the individual refinement locations. We have manually investigated all of the program locations where refinement queries are issued and categorized them into one of the following categories:

(1) Intraprocedural correlated read/write pattern
(2) Interprocedural correlated read/write pattern
(3) No correlated read/write pattern
(4) Dataflow through free variable

In addition, we make note of two other meaningful characteristics of refinement locations:

a. Transformation of the read, e.g. `target[key] = f(source[key])`
b. Conditional property write

The numbered categories 1, 2, and 3 indicate whether a correlated read/write is present and, if so, whether it spans a procedure boundary. Category 4 indicates that the memory location being refined is a free variable in the function where refinement is triggered.

The lettered categories describe extra characteristics about the property read/write patterns: "a" patterns transform the read property value before writing it, and "b" patterns perform the property write conditionally only when a predicate is satisfied.

A particular refinement location belongs to one of the numbered categories and any number of the lettered categories.[10]

The results of the investigation are shown for Underscore, Lodash3, Lodash4, Prototype, Scriptaculous, and jQuery in tables 4, 5, 6, 7, 8, and 9 respectively.

The tables each have the same structure and provide the following information: "Ref. loc" is the source location in the program where refinement queries are issued, written as "line:column"; "Queries" is the total number of refinement queries issued; "Success" is the percentage of queries where the value refiner produces a result more precise than the base analysis state for the requested memory address; "Avg. time" is the average time spent by the value refiner on each query; "Avg. locs" is the average number of program locations visited per refinement query; "Inter." is the percentage of the queries where the value refiner visits multiple functions; "PNI" is the percentage of queries where the value refiner applies property name inference (see Section 5.3); "Test cases" is the number of test cases that issue refinement queries; and "Category" is a categorization of the refinement location as defined above.

In total, the refiner fails to provide precise results at seven locations, where four of them do not belong to a correlated property read/write. However, 24 of the 35 refinement locations are in correlated property read/write pairs (i.e. categories 1 and 2), indicating that such patterns are often a reason for loss of precision by the base analysis. Of the 24 correlated property read/writes, we see that 19 are intraprocedural and 5 are interprocedural.

[10]With the exception of the locations categorized "2 or 3", where the categorization depends on the invocation context of the containing function, as described below.

Table 4. Statistics for all refinement queries issued in the 182 Underscore test cases.

Ref. loc	Queries	Success (%)	Avg. time (ms)	Avg. locs	Inter. (%)	PNI (%)	Test cases	Category
1492:18	48412	100.0	2	5	0.0	100.0	182	1
1493:7	114	100.0	9	9	0.0	0.0	25	1
22:9	24	87.5	30	13	4.8	95.2	1	1
1037:38	56	85.7	13	39	100.0	100.0	2	1, b
108:54	141	100.0	1	5	0.0	100.0	11	1

Table 5. Statistics for all refinement queries issued in the 176 Lodash3 test cases.

Ref. loc	Queries	Success (%)	Avg. time (ms)	Avg. locs	Inter. (%)	PNI (%)	Test cases	Category
10682:7	16	100.0	4	18	100.0	0.0	8	4
2358:11	2	0.0	-	-	0	0	1	3
3739:11	2	0.0	-	-	0	0	1	2, b
10540:11	32800	100.0	8	15	100.0	100.0	176	2
10084:9	49781	100.0	3	7	0.0	100.0	176	1
3720:11	217	99.1	25	5	0.0	100.0	2	1
2331:11	41	100.0	50	67	100.0	4.9	1	1, a, b
10086:11	28	100.0	7	18	100.0	0.0	8	4
2388:9	6	66.7	39	75	100.0	0.0	1	1, a
2386:9	2	0.0	-	-	0	0	1	1, a
1547:11	752	100.0	14	40	100.0	100.0	2	1, a, b
2827:9	2	0.0	-	-	0	0	1	3

Note that, for the two locations in Table 6 categorized as “2 or 3”, the categorization depends on the invocation context of the function containing the refinement location. In our experiments, most of the refinement queries at locations 2002:7 and 2573:9 are issued in correlated read/write contexts (i.e. in category 2).

The tables also show that the refiner almost always provides precise results when analyzing correlated property read/write patterns, whether those patterns are intraprocedural or interprocedural. We also see that property name inference is used in most cases. As such, the refinements also depend on the precision of the current analysis state; for instance, at Lodash3 location 2388:9, we see that they succeed for most but not all refinements. The failing refinements are issued in the same test case as the location 2358:11 refinement. Since the value refiner fails to precisely refine at 2358:11, imprecision is introduced to the analysis state which is propagated to 2388:9, meaning that the base analysis state is not precise enough to perform property name inference. The refinement at line 2386:9 fails for the same reason. For some of the other cases where refinements is sometimes but not always successful, it is typically because the refinements at that location are highly variable. For instance, category “a” and “b” locations often require reasoning about a callback function that is provided in the test case, so successful refinement depends heavily on whether the refiner is able to reason precisely about that function.

The tables also show that most successful refinements happen quite close to the refinement location. The refinement that visited the most locations visited 161 locations on average, and most refinements involved visiting less than 50 locations. This demonstrates that the loss of precision is often proximate to the critical imprecise property write. However, even though most refinements visit relatively few locations, 18 of the 35 refinement locations nonetheless require interprocedural reasoning.

Table 6. Statistics for all refinement queries issued in the 306 Lodash4 test cases.

Ref. loc	Queries	Success (%)	Avg. time (ms)	Avg. locs	Inter. (%)	PNI (%)	Test cases	Category
15702:9	291417	100.0	5	7	0.0	100.0	306	1
16840:11	99498	100.0	8	17	100.0	100.0	306	2
2573:9	500	89.6	361	97	100.0	97.1	14	2 or 3
15704:11	84	100.0	51	14	0.0	0.0	10	4
16983:7	45	97.8	32	40	100.0	0.0	12	4
2002:7	1209	94.7	1229	161	100.0	0.0	16	2 or 3
12808:13	164	100.0	4	5	0.0	100.0	1	1

Table 7. Statistics for all refinement queries issued in the 6 Prototype test cases.

Ref. loc	Queries	Success (%)	Avg. time (ms)	Avg. locs	Inter. (%)	PNI (%)	Test cases	Category
145:7	1102	100.0	13	40	46.8	100.0	6	1
7155:5	13	100.0	3	5	100.0	0.0	6	4
7157:5	4	100.0	4	5	100.0	0.0	3	4
3380:9	10	100.0	3	5	100.0	0.0	3	1, a

Table 8. Statistics for all refinement queries issued in the Scriptaculous test case.

Ref. loc	Queries	Success (%)	Avg. time (ms)	Avg. locs	Inter. (%)	PNI (%)	Test cases	Category
145:7	597	100.0	13	37	41.9	100.0	1	1
7155:5	4	100.0	4	5	100.0	0.0	1	4

Table 9. Statistics for all refinement queries issued in the JQuery test cases.

Ref. loc	Queries	Success (%)	Avg. time (ms)	Avg. locs	Inter. (%)	PNI (%)	Test cases	Category
368:6	8	87.5	85	26	0.0	0.0	8	1
8369:4	5	0.0	-	-	0	0	5	1, a
5188:4	2	0.0	-	-	100.0	0.0	1	3
5200:4	63	100.0	5	5	0.0	100.0	1	1
3650:3	2	0.0	-	-	0	0	1	3

ACKNOWLEDGMENTS

We are grateful to Esben Andreasen for his contributions to the early phases of this research. This work was supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No 647544) and in part by NSF under grants CCF-1619282 and CCF-1055066 and by DARPA under agreement number FA8750-14-2-0263.

REFERENCES

Roberto Amadini, Alexander Jordan, Graeme Gange, François Gauthier, Peter Schachte, Harald Søndergaard, Peter J. Stuckey, and Chenyi Zhang. 2017. Combining String Abstract Domains for JavaScript Analysis: An Evaluation. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017 (Lecture Notes in Computer Science)*, Vol. 10205. 41–57.

Esben Andreasen and Anders Møller. 2014. Determinacy in Static Analysis for jQuery. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014*. ACM, 17–31.

Esben Sparre Andreasen, Anders Møller, and Benjamin Barslev Nielsen. 2017. Systematic Approaches for Increasing Soundness and Precision of Static Analyzers. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis, SOAP@PLDI 2017*. 31–36.

Thomas Ball, Orna Kupferman, and Greta Yorsh. 2005. Abstraction for Falsification. In *Computer Aided Verification, 17th International Conference, CAV 2005 (Lecture Notes in Computer Science)*, Vol. 3576. Springer, 67–81.

Thomas Ball and Sriram K. Rajamani. 2001. Automatically Validating Temporal Safety Properties of Interfaces. In *Model Checking Software, 8th International SPIN Workshop, 2001 (Lecture Notes in Computer Science)*, Vol. 2057. Springer, 103–122.

Sam Blackshear, Bor-Yuh Evan Chang, and Manu Sridharan. 2013. Thresher: Precise Refutations for Heap Reachability. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*. 275–286.

Satish Chandra, Stephen J. Fink, and Manu Sridharan. 2009. Snugglebug: A Powerful Approach to Weakest Preconditions. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009*. ACM, 363–374.

Bor-Yuh Evan Chang and K. Rustan M. Leino. 2005. Abstract Interpretation with Alien Expressions and Heap Structures. In *Verification, Model Checking, and Abstract Interpretation, 6th International Conference, VMCAI 2005*. 147–163.

David R. Chase, Mark N. Wegman, and F. Kenneth Zadeck. 1990. Analysis of Pointers and Structures. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation, PLDI 1990*. ACM, 296–310.

Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2000. Counterexample-Guided Abstraction Refinement. In *Computer Aided Verification, 12th International Conference, CAV 2000 (Lecture Notes in Computer Science)*, Vol. 1855. Springer, 154–169.

Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, POPL 1977*. ACM, 238–252.

Patrick Cousot and Radhia Cousot. 1979. Systematic Design of Program Analysis Frameworks. In *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages POPL 1979*. 269–282.

Patrick Cousot and Radhia Cousot. 1992. Abstract Interpretation Frameworks. *J. Log. Comput.* 2, 4 (1992), 511–547.

Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2006. Combination of Abstractions in the ASTRÉE Static Analyzer. In *Advances in Computer Science - ASIAN 2006. Secure Software and Related Issues, 11th Asian Computing Science Conference (Lecture Notes in Computer Science)*, Vol. 4435. Springer, 272–300.

Patrick Cousot, Radhia Cousot, and Francesco Logozzo. 2011. Precondition Inference from Intermittent Assertions and Application to Contracts on Collections. In *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011*. 150–168.

Arlen Cox, Bor-Yuh Evan Chang, and Xavier Rival. 2014. Automatic Analysis of Open Objects in Dynamic Language Programs. In *Static Analysis - 21st International Symposium, SAS 2014*. 134–150.

Kyle Dewey, Vineeth Kashyap, and Ben Hardekopf. 2015. A Parallel Abstract Interpreter for JavaScript. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2015*. IEEE Computer Society, 34–45.

Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. 2002. Extended Static Checking for Java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 234–245.

Philippa Gardner, Sergio Maffeis, and Gareth David Smith. 2012. Towards a Program Logic for JavaScript. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012*. 31–44.

Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. 2010. The Essence of JavaScript. In *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Proceedings*. Springer, 126–150.

Bhargav S. Gulavani and Sriram K. Rajamani. 2006. Counterexample Driven Refinement for Abstract Interpretation. In *Tools and Algorithms for the Construction and Analysis of Systems, 12th International Conference, TACAS 2006 (Lecture Notes in Computer Science)*, Vol. 3920. Springer, 474–488.

Samuel Z. Guyer and Calvin Lin. 2005. Error Checking with Client-Driven Pointer Analysis. *Sci. Comput. Program.* 58, 1-2 (2005), 83–114.

Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. 2002. Lazy Abstraction. In *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 58–70.

IBM Research. 2018. *T.J. Watson Libraries for Analysis (WALA)*.

Samin S. Ishtiaq and Peter W. O'Hearn. 2001. BI as an Assertion Language for Mutable Data Structures. In *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 14–26.

Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2009. Type Analysis for JavaScript. In *Static Analysis, 16th International Symposium, SAS 2009 (Lecture Notes in Computer Science)*, Vol. 5673. Springer, 238–255.

John B. Kam and Jeffrey D. Ullman. 1977. Monotone Data Flow Analysis Frameworks. *Acta Inf.* 7 (1977), 305–317.

Vineeth Kashyap, Kyle Dewey, Ethan A. Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. 2014. JSAI: A Static Analysis Platform for JavaScript. In *Proc. 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 121–132.

Gary A. Kildall. 1973. A Unified Approach to Global Program Optimization. In *Conference Record of the ACM Symposium on Principles of Programming Languages, POPL 1973.* ACM Press, 194–206.

Yoonseok Ko, Xavier Rival, and Sukyoung Ryu. 2017. Weakly Sensitive Analysis for Unbounded Iteration over JavaScript Objects. In *Programming Languages and Systems - 15th Asian Symposium, APLAS 2017.* 148–168.

Yoonseok Ko, Xavier Rival, and Sukyoung Ryu. 2019. Weakly Sensitive Analysis for JavaScript Object-Manipulating Programs. *Softw., Pract. Exper.* 49, 5 (2019), 840–884.

Hongki Lee, Sooncheol Won, Joonho Jin, Junhee Cho, and Sukyoung Ryu. 2012. SAFE: Formal Specification and Implementation of a Scalable Analysis Framework for ECMAScript. In *Proc. International Workshop on Foundations of Object Oriented Languages (FOOL 2012).*

Sorin Lerner, David Grove, and Craig Chambers. 2002. Composing Dataflow Analyses and Transformations. In *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* 270–282.

Percy Liang and Mayur Naik. 2011. Scaling Abstraction Refinement via Pruning. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011.* 590–601.

Magnus Madsen and Esben Andreasen. 2014. String Analysis for Dynamic Field Access. In *Proc. 23rd International Conference on Compiler Construction (Lecture Notes in Computer Science)*, Vol. 8409. Springer.

Roman Manevich, Manu Sridharan, Stephen Adams, Manuvir Das, and Zhe Yang. 2004. PSE: Explaining Program Failures via Postmortem Static Analysis. In *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2004.* ACM, 63–72.

Hakjoo Oh, Wonchan Lee, Kihong Heo, Hongseok Yang, and Kwangkeun Yi. 2016. Selective X-Sensitive Analysis Guided by Impact Pre-Analysis. *ACM Trans. Program. Lang. Syst.* 38, 2 (2016), 6:1–6:45.

Changhee Park, Hyeonseung Im, and Sukyoung Ryu. 2016. Precise and Scalable Static Analysis of jQuery using a Regular Expression Domain. In *Proceedings of the 12th Symposium on Dynamic Languages, DLS 2016.* ACM, 25–36.

Changhee Park and Sukyoung Ryu. 2015. Scalable and Precise Static Analysis of JavaScript Applications via Loop-Sensitivity. In *Proc. 29th European Conference on Object-Oriented Programming.* 735–756.

Xavier Rival and Laurent Mauborgne. 2007. The Trace Partitioning Abstract Domain. *ACM Trans. Program. Lang. Syst.* 29, 5 (2007), 26.

José Fragoso Santos, Petar Maksimovic, Daiva Naudziuniene, Thomas Wood, and Philippa Gardner. 2018. JaVerT: JavaScript Verification Toolchain. *PACMPL* 2, POPL (2018), 50:1–50:33.

José Fragoso Santos, Petar Maksimovic, Gabriela Sampaio, and Philippa Gardner. 2019. JaVerT 2.0: Compositional Symbolic Execution for JavaScript. *PACMPL* 3, POPL (2019), 66:1–66:31.

Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. 2016. Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java. In *30th European Conference on Object-Oriented Programming, ECOOP 2016.* 22:1–22:26.

Manu Sridharan and Rastislav Bodík. 2006. Refinement-Based Context-Sensitive Points-To Analysis for Java. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, PLDI 2006.* 387–400.

Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip. 2012. Correlation Tracking for Points-To Analysis of JavaScript. In *Proc. 26th European Conference on Object-Oriented Programming.*

Antoine Toubhans, Bor-Yuh Evan Chang, and Xavier Rival. 2013. Reduced Product Combination of Abstract Domains for Shapes. In *Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013.* 375–395.

Shiyi Wei, Omer Tripp, Barbara G. Ryder, and Julian Dolby. 2016. Revamping JavaScript Static Analysis via Localization and Remediation of Root Causes of Imprecision. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016.* ACM, 487–498.