Value Partitioning: A Lightweight Approach to Relational Static Analysis for JavaScript

Benjamin Barslev Nielsen
Aarhus University, Denmark
barslev@cs.au.dk

Anders Møller
Aarhus University, Denmark
amoeller@cs.au.dk

Abstract

In static analysis of modern JavaScript libraries, relational analysis at key locations is critical to provide sound and useful results. Prior work addresses this challenge by the use of various forms of trace partitioning and syntactic patterns, which is fragile and does not scale well, or by incorporating complex backwards analysis. In this paper, we propose a new lightweight variant of trace partitioning named value partitioning that refines individual abstract values instead of entire abstract states. We describe how this approach can effectively capture important relational properties involving dynamic property accesses, functions with free variables, and predicate functions. Furthermore, we extend an existing JavaScript analyzer with value partitioning and demonstrate experimentally that it is a simple, precise, and efficient alternative to the existing approaches for analyzing widely used JavaScript libraries.

2012 ACM Subject Classification Theory of computation → Program analysis

Keywords and phrases JavaScript, dataflow analysis, abstract interpretation

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2020.16

Funding 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).

1 Introduction

JavaScript programs are challenging to analyze statically due to the dynamic nature of the language. One of the main obstacles is the presence of *dynamic property access* operations that allow objects to be manipulated using object property names that are dynamically computed strings. A typical pattern that has received much attention is *correlated read/write pairs* [25], a simple variant of which looks as follows:

```
t = x[p];   ...   y[p] = t;
```

At run-time, this code copies a property whose name is the value of `p` from the `x` object to the `y` object. If the static analysis does not know precisely the string value of `p`, then the properties of `x` will be mixed together in `y`. Experience with analyzers such as WALA [25, 24, 28], SAFE [17, 22], JSAI [13], and TAJS [11, 2, 26] has shown that when analyzing real-world JavaScript code, including jQuery, Lodash, Underscore and other widely used libraries, such situations often cause an avalanche of spurious dataflow that makes the analysis results useless. If, for example, `x` is the object `{m1: f1, m2: f2, ..., m10: f10}` where `f1`, `f2`, . . . , `f10` are functions, then any subsequent function call, for example `y.m3(...)`, will be treated by the analysis as a call to any of the 10 functions.

Several analysis techniques have been proposed to address this challenge. The techniques based on correlation tracking [25], static/dynamic determinacy [24, 2], and loop sensitivity [22] aim to increase precision by the use of context sensitivity or loop unrolling to ensure that

34th European Conference on Object-Oriented Programming (ECOOP 2020).
Editors: Robert Hirschfeld and Tobias Pape; Article No. 16; pp. 16:1–16:28
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

the analysis has precise information about `p` in the example above. Although this approach works well in many cases, the aggressive use of context sensitivity or loop unrolling can be expensive on analysis time. Even more importantly, it falls short when `p` is not determinate (i.e., when its value is not fixed even when the call context is known).

An important step forward is the approach used in the CompAbs analyzer, which is built on SAFE [16]: Even if `p` is imprecise, the loss of precision at the property write operation can be avoided by applying trace partitioning [23] at the property read operation, based on which properties exist on `x`. Intuitively, it is often not necessary to have precise information about `p`; instead we can refine the current abstract state into a collection of more precise partitions, one for each of the 10 properties of `x` (plus one extra for the case where `p` is none of those strings, but let us ignore that for now), and after the property write operation merge them again. (The same idea was used earlier in TAJS, but only at `for-in` loops, not at dynamic property reads [2].) This approach, however, also has drawbacks. Trace partitioning is expensive, so it must be used scarcely: in the example, the code between the dynamic property read and the dynamic property write is essentially analyzed 10 times. For this reason, CompAbs relies on a syntactic pre-analysis to recognize different kinds of correlated read/write pairs for guiding the creation and merging of partitions.

Recent work [26] has shown that the syntactic pre-analysis approach of CompAbs is too fragile, for example, it is incapable of analyzing the Lodash library (see Section 2), and demand-driven value refinement has been proposed as an alternative. Instead of relying on context sensitivity, loop unrolling, or trace partitioning, that approach applies, during the analysis when encountering a dynamic property write operation with an imprecise property name, a separate backwards analysis to regain the relation between the property name and the value to be written. Although demand-driven value refinement has been shown to work quite well in practice, building a backwards analysis for the full JavaScript language and its standard library is a major endeavor, so developing simpler alternatives is desirable.

Our approach builds upon the observation from CompAbs that sufficient precision can be obtained using trace partitioning based on the properties of the object being read. Our key insight is that we do not need to partition the entire abstract state as done by CompAbs: It suffices to only partition the abstract values for the property name `p` and the value being read `x[p]` in the above example. This means that instead of analyzing the code 10 times, we only analyze it once, but using partitioned abstract values that retain the correlation between `p` and `x[p]`. The partitioned abstract values are introduced at `t = x[p]` and used at `y[p] = t` by means of specialized transfer functions. We refer to this variant of trace partitioning as *value partitioning*. Since partitioning individual abstract values does not increase the analysis complexity as much as partitioning entire states, it becomes feasible to apply value partitioning more extensively, at every dynamic property read where the property name is imprecise, thereby obviating the need for the syntactic pre-analysis.

In this paper we present a theoretical framework for value partitioning, together with three instantiations: *property-name partitioning* (which is the one used in the example above), *free-variable partitioning* (to improve precision for free variables of closures), and *type partitioning* (to improve precision for predicate functions). Additionally, we extend the static analyzer TAJS with all three kinds of value partitioning and demonstrate that the approach is effective for analyzing popular JavaScript libraries. Value partitioning is a lightweight alternative to the existing approaches to relational analysis for JavaScript: Compared to CompAbs-style trace partitioning it avoids many redundant computations caused by similarities between different partitions, and compared to demand-driven value refinement it avoids the need for creating a separate backwards analysis.

```
 1 function mixin(object, source) {
 2   baseFor(source, function (func, methodName) {
 3     if (!isFunction(func))
 4       return;
 5     object[methodName] = func;
 6     if (isFunction(object))
 7       object.prototype[methodName] = function() {
 8         ...
 9         func.apply(...);
10       }
11   });
12 }
13
14 function baseFor(source, iteratee) {
15   Object.keys(source).forEach(function (key) {
16     iteratee(source[key], key);
17   });
18 }
19
20 // usage of mixin during initialization
21 mixin(lodash, lodash);
```

Figure 1 Motivating example based on code from the Lodash library.

In summary our contributions are:

- Value partitioning: a general static analysis technique that is capable of reasoning about relations between abstract values.
- Three instantiations of value partitioning, which tackle different challenges in static analysis for JavaScript, each involving relational properties:
  - property-name partitioning: relations between dynamically computed object property names and values;
  - free-variable partitioning: relations between functions and their free variables; and
  - type partitioning: relations between arguments and return values of predicate functions.
- Experimental results: We show that value partitioning makes TAJS more precise than CompAbs [16] for several real-world JavaScript libraries, including Lodash, which is the most widely used library. The resulting precision is comparable to (and in case of the Lodash4 benchmark group substantially higher than) that of demand-driven value refinement [26], without the need for a separate backwards analysis.

2 Motivating Example and Overview

Figure 1 shows a small code example based on Lodash (version 4.17.10), which is the most depended-upon of all npm packages.[1] Lines 1–12 define the function `mixin`, which copies all function properties from `source` to `object`. If `object` is a function, a new function (which

[1] Lodash (`https://lodash.com/`) has more than 115 000 dependents in npm and more than 27 million weekly downloads as of May 2020.

on invocation calls the function to be copied) is also copied to object.prototype, such that instantiations of object (using the keyword new) also will have these functions. In line 21, which is executed during the initialization of Lodash, mixin is called with the library object as both arguments. The function mixin uses a helper function baseFor defined in lines 14–18. It is called with source and a callback function defined in lines 2–11. The baseFor function then gets all the object property names from the source object using Object.keys, and the callback function is called (line 16) for each property name and corresponding property value. Line 3 checks whether func is a function. If so, the function is copied to object[methodName] in line 5. Note that func actually is the value source[methodName]. Line 6 checks whether object is also a function and if so, a new function is declared and written to object.prototype[methodName] in line 7. When invoked, that new function calls func using func.apply(...) in line 9.

Such complex code is not unusual in modern JavaScript libraries. For a static analysis reasoning about the dataflow in this code, the correlation between methodName and func is critical. An analysis that loses track of this correlation will mix together all the properties of the library object lodash when analyzing the call mixin(lodash, lodash) in line 21. As a consequence, if the program being analyzed contains a call to, for example, lodash.map, that will be treated by the analysis as a call to any of Lodash's more than 100 different functions, not only the actual map function, thereby triggering an avalanche of spurious dataflow.

Existing approaches

Existing JavaScript analyzers do not have precise information about the value of key in line 16, for various different reasons. (Most importantly, Object.keys produces an array of property names in unspecified order.) Previous work has suggested two approaches to analyze such code precisely even when key is imprecise. The CompAbs [16] approach uses trace partitioning guided by syntactic patterns. If trace partitioning is used at the dynamic property read operation in line 16, the abstract state is partitioned into a set of refined abstract states corresponding to the properties of the source object. This way the value of key is precise in each of those states, and the call in line 16 is analyzed separately for each of them. Trace partitioning, however, is expensive, so CompAbs limits the use of trace partitioning according to certain syntactic patterns. At this specific dynamic property operation, CompAbs chooses not to apply trace partitioning and fails to detect that the relation between methodName and func is important.

The second approach is demand-driven value refinement [26], which can analyze the example code with sufficient precision to avoid mixing together the Lodash functions. With this approach, the analysis detects imprecision at the dynamic property write in line 5: methodName is an imprecise string and func can be many different functions. It then queries a backwards abstract interpreter asking for the possible value of methodName for each of the functions. The backwards analysis returns a precise property name for each function and thereby enables the dynamic property write operation to be modeled precisely. For the dynamic property write in line 7, the function defined in lines 7–10 is written to all properties of object.prototype, but the abstract value being written is augmented, such that the value of methodName remains precise. When reading func in line 9, the backwards analysis is queried to get the value of func relative to the value of methodName, thereby retrieving a precise value for func. This ensures the desired precision, but the approach requires a complicated backwards analysis.

Value partitioning

We will now informally explain how value partitioning can provide similar precision as demand-driven value refinement, but without the need for a backwards abstract interpreter. With traditional trace partitioning, as used by, for example, CompAbs, the analysis can track multiple abstract states for each program point, such that the different abstract states cover different assumptions about the execution paths that lead to that point. (Correlation tracking [25], determinacy-based analysis [24, 2], and loop sensitivity [22] can also be viewed as variations of trace partitioning.) The key idea behind value partitioning is that we can obtain a similar effect as trace partitioning by instead performing the partitioning at the level of individual abstract values. In principle, the resulting abstract domain is isomorphic to a traditional trace partitioning domain, but this approach provides more flexibility for using different kinds of partitioning for different parts of the abstract states. This general idea can be instantiated in multiple ways to track different kinds of relational properties. We next describe three instantiations that enable precise analysis of challenging JavaScript code, including the Lodash example.

Property name partitioning

One instantiation is property name partitioning, which performs partitioning at dynamic property reads, similar to the CompAbs technique, but on abstract values instead of abstract states. To illustrate this mechanism by example, consider the read operation in line 16 and the correlated write operation in line 5. Assume for simplicity that the `source` object has only two properties, `{map: f1, trim: f2}` where `f1` and `f2` are functions, and `methodName` is an abstract value that overapproximates all valid property names. When reading `source[methodName]`, an analysis without value partitioning will read all the properties of `source`. When using value partitioning, we instead partition this value according to the property names of `source`, meaning that we obtain a value $[t_1 \mapsto \texttt{f1}, t_2 \mapsto \texttt{f2}, t_3 \mapsto \texttt{undefined}]$ where t_1, t_2, and t_3 represent different partitions.[2] Intuitively, t_1 represents the execution traces where the property name being read is `map`, t_2 similarly represents traces where the property name being read is `trim`, and t_3 represents all other traces. We similarly write the partitioned value $[t_1 \mapsto \texttt{"map"}, t_2 \mapsto \texttt{"trim"}, t_3 \mapsto \mathsf{AnyString}]$ to `methodName`.[3] In this way, the resulting abstract state retains the correlation between the values of `methodName` and `source[methodName]`.

Later the analysis reaches the write operation `object[methodName] = func`, with an abstract state where `methodName` is $[t_1 \mapsto \texttt{"map"}, t_2 \mapsto \texttt{"trim"}, t_3 \mapsto \mathsf{AnyString}]$ and `func` is $[t_1 \mapsto \texttt{f1}, t_2 \mapsto \texttt{f2}, t_3 \mapsto \texttt{undefined}]$. Since the property name and the value to be written have the same partitions, we can perform the dynamic property write separately for each partition, meaning that `f1` is written to the `map` property, and analogously for the other two partitions, thereby avoiding mixing together the properties.

Since the partitioning is performed at the value level, unlike traditional trace partitioning we do not need any extra call contexts to the callback function defined in line 2, so the overhead of value partitioning is negligible, even when the correlated read/write pairs span multiple functions. For this reason, we can apply property name partitioning at all dynamic property reads where the property name is imprecise, without the use of syntactic patterns.

[2] In JavaScript, reading an absent property yields the special value `undefined`.

[3] $\mathsf{AnyString}$ is an abstract value that represents any string. In practice we instead use a slightly more precise abstract value representing $\mathsf{AnyString}\backslash\{\texttt{"map"}, \texttt{"trim"}\}$.

Free variable partitioning

A second instantiation of value partitioning is for handling free variables more precisely. In the example, this is useful for `func` in line 9, which is a free variable in the function defined in lines 7–10. At that function definition, we partition both the resulting abstract function value ℓ and the abstract value of `func` according to the existing partitioning of `func`, intuitively to be able to distinguish functions created with different values of the free variable. This means that the function value being written at the dynamic property write in line 7 is $[t_1 \mapsto \ell_{t'_1}, t_2 \mapsto \ell_{t'_2}, t_3 \mapsto \ell_{t'_3}]$ where $\ell_{t'_1}$ represents the function created at a point where `func` is `f1` (i.e., that point is at the end of a t_1 trace), and similarly for the other partitions. At the same time, the value of `func` becomes $[t_1 \mapsto \texttt{f1}, t_2 \mapsto \texttt{f2}, t_3 \mapsto \texttt{undefined}, t'_1 \mapsto \texttt{f1}, t'_2 \mapsto \texttt{f2}, t'_3 \mapsto \texttt{undefined}]$ where the three new partitions t'_1, t'_2, and t'_3 denote the new partitioning we have made (one abstract value can thus have multiple partitionings simultaneously). Using the property name partitioning mechanism described above, at the dynamic property write in line 7, $\ell_{t'_1}$ is written to the `map` property of `object.prototype`, and similarly for the other properties.

We can exploit the free variable partitioning information when the function is later called. Assume the analysis encounters a call to the `map` method. The abstract value of `lodash.prototype.map` is then $\ell_{t'_1}$. We now use t'_1 as a context in ordinary context sensitive analysis of the function, so that when reaching `func` in line 9, it suffices to consider only the t'_1 partition of `func`, which yields the precise value `f1`, so again, we successfully avoided mixing together the properties.

Type partitioning

The above two uses of value partitioning are sufficient for analyzing the motivating example without critical precision losses, but we can make the analysis even more precise using a third variant. The function named `isFunction` used in the branch condition in line 6 is a typical example of a *predicate function*, i.e., a one-parameter function that returns a boolean, in this case testing whether the value passed in is a function. Assume the abstract value of the argument `object` is `fun1`|`obj2`, meaning that it represents either a function `fun1` or a non-function object `obj2`. With a simple analysis, the abstract return value and hence the branch condition is Bool representing any boolean value, so the analysis does not know that `object` cannot be `obj2` inside the branch. This causes the analysis to spuriously raise a type error when writing to `object.prototype` in line 7.

Type partitioning avoids that imprecision as follows. Type partitioning is triggered at any call to a function with one argument, and partitions that argument according to its types. In this case, the value of `object` is partitioned into $[a \mapsto \texttt{fun1}, b \mapsto \texttt{obj2}]$. The result value from `isFunction` then becomes $[a \mapsto \mathsf{true}, b \mapsto \mathsf{false}]$, which we can exploit using ordinary control sensitivity [10] (also called type refinement [14]) at the “true” branch such that `object` in line 7 will only be `fun1` and not `obj2`.

Overview

In Section 3 we give a brief introduction to the analysis domain of TAJS. Section 4 explains the general value partitioning mechanism, and Section 5 details the three instantiations: property name partitioning, free variable partitioning, and type partitioning. Section 6 describes our experimental evaluation, and Section 7 discusses related work.

$r_1[r_2] \leftarrow r_3$:	Writes r_3 to the property named r_2 of the object r_1
$r_1 \leftarrow r_2[r_3]$:	Reads the property named r_3 of the object r_2 to r_1
$r_1 \leftarrow x$:	Reads the value of the variable x to r_1
$x \leftarrow r_1$:	Writes r_1 to the variable x
$r_1 \leftarrow c$:	Assigns the constant c to r_1
$r_1 \leftarrow \mathsf{function}(x)\{\cdots\}$:	Creates a closure for the function and stores it in r_1
$\mathsf{if}(r_1)$:	Conditionally propagates dataflow (to model `if` and `while`)
$r_1 \leftarrow r_2(r_3)$:	Calls the function r_2 with argument r_3 and stores the result in r_1
$r_1 \leftarrow r_2 \oplus r_3$:	Computes the binary operation $r_2 \oplus r_3$ and stores the result in r_1

Figure 2 The main flow graph instructions in TAJS.

$$\begin{aligned} n \in N &: \mathit{Nodes} \\ c \in C &: \mathit{Contexts} \\ p \in P &: \mathit{Property\ names} \\ \ell \in L = N \times C &: \mathit{Locations} \end{aligned} \qquad \begin{aligned} X \in \mathit{AnalysisLattice} &= L \to \mathit{State} \\ \sigma \in \mathit{State} &= (L \to \mathit{Obj}) \times \mathit{Registers} \\ o \in \mathit{Obj} &= P \to \mathit{Value} \\ r \in \mathit{Registers} &= R \to \mathit{Value} \\ v \in \mathit{Value} &= \mathit{Prim} \times \mathcal{P}(L) \end{aligned}$$

Figure 3 Simplified abstract domain.

3 Background: The TAJS Analyzer

In this section we give a brief introduction to a heavily simplified version of the analysis domain and program representation used in TAJS [11, 2], which lays the foundation for our extensions in the following sections.

TAJS is an open-source dataflow analysis tool for JavaScript built as a monotone framework [12]. A JavaScript program is represented as a control flow graph for each function, with nodes representing primitive instructions of the different kinds listed in Figure 2. Each instruction operates on registers, which can be thought of as special local variables. For simplicity, we ignore `this` and receiver objects at calls, and we assume all functions have only one parameter. As an example, the single JavaScript statement `y[p] = x[p]` is represented as six flow graph nodes as shown in Figure 4.

The components of the abstract domain are summarized in Figure 3. A *location* is a pair of a node and a context. The contexts allow for context sensitivity (using the context-sensitivity strategy described by Andreasen and Møller [2]). The main abstract domain, *AnalysisLattice*, is a lattice that maps locations to abstract states, where each state contains abstract values of object properties and registers. Objects are modeled using context-sensitive allocation-site abstraction [6, 20], so abstract object addresses are simply locations.[4] Functions are special kinds of objects. Abstract values are modeled using a product of a constant-propagation lattice [15] named *Prim* of primitive values (strings, numbers, etc.) and a powerset lattice of object addresses.

The analysis is control sensitive by pruning infeasible dataflow at if nodes. This includes not only eliminating flow along unreachable branches, for example when a branch condition is definitely false [27], but also filtering abstract values based on the branch condition [10, 14].

[4] TAJS models absence/presence of object properties and uses two artificial properties DefaultNumeric and DefaultOther to model properties with unknown numeric/non-numeric names; we ignore that here.

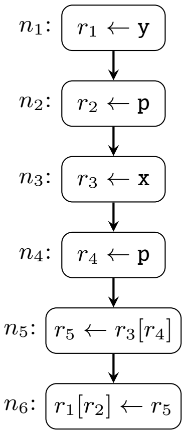
n_1: $r_1 \leftarrow$ y
n_2: $r_2 \leftarrow$ p
n_3: $r_3 \leftarrow$ x
n_4: $r_4 \leftarrow$ p
n_5: $r_5 \leftarrow r_3[r_4]$
n_6: $r_1[r_2] \leftarrow r_5$

Figure 4 Fragment of a control flow graph, for the single statement `y[p] = x[p]`.

As an example, the JavaScript code `if(z)` is represented by two primitive instructions, $r_6 \leftarrow$ `z` and $\mathsf{if}(r_6)$. In the "true" branch, not only r_6 but also `z` must have the value `true`.[5] To track the connection between r_6 and `z`, a simple intraprocedural must-equals analysis is performed alongside the main dataflow analysis. We leverage this mechanism in Section 5, for example to obtain the information that r_2, r_4, and `p` must have the same value at the property read operation in Figure 4 (unless a property accessor changes `p`). To keep Figure 3 simple, we omit the must-equals information in the description of the *State* lattice.

In the following sections, with a slight abuse of notation we let $\sigma(r)$ denote the value of register r in state σ, and similarly, $\sigma(\mathtt{x})$ denotes the value of variable `x`. Also, we use the notation $\sigma(r) := \ldots$ to describe the operation of writing a given value to register r and also to the variables and registers that are equal to r according to the must-equals information. If $\ell \in L$ is a location representing an object address, we sometimes write ℓ for the abstract value $(\bot, \{\ell\}) \in \mathit{Value}$. Similarly, for abstract values that represent primitive values only, we omit the location sets, for example, `"foo"` denotes the abstract value $(\texttt{"foo"}, \emptyset) \in \mathit{Value}$.

We omit many details of TAJS, including the definitions of the concretizations of the lattice elements, the definitions of the transfer functions for the different instructions, how values of variables are being stored in special activation objects, and how a call graph is built during the analysis. Analyzing full JavaScript also requires reasoning about prototypes, scope chains, implicit type conversions, exceptions, the standard library, property accessors (getters and setters), and much more. It suffices to know that the resulting abstract states soundly overapproximate the possible program behavior [7].

A *trace* is a concrete execution of the program expressed as a finite sequence of pairs (ℓ, γ) where ℓ is a location and γ is a concrete state, starting at the program entry point with the initial call context in an empty state. The semantics of a program is defined as a set of traces. The *collecting semantics* is the program semantics projected onto the program locations: Given a location ℓ, the collecting semantics for ℓ, denoted $[\![\ell]\!]$, is the set of states that appear at ℓ in the set of traces defined by the program semantics. The analysis result is thus a lattice element $X \in \mathit{AnalysisLattice}$ such that $[\![\ell]\!]$ is a subset of the concretization of $X(\ell)$ for all locations $\ell \in L$.

[5] In actual JavaScript, the value must be *truthy*, which also includes nonempty strings, nonzero numbers, and objects.

$$
\begin{aligned}
t \in T &\ : \mathit{Partition\ tokens} \\
o \in \mathit{Obj} &= P \to \mathit{PartitionedValue} \\
r \in \mathit{Registers} &= R \to \mathit{PartitionedValue} \\
pv \in \mathit{PartitionedValue} &= T \hookrightarrow \mathit{Value}
\end{aligned}
$$

Figure 5 Extension of the abstract domain for value partitioning.

4 Value Partitioning

To prepare the analysis for value partitioning, we introduce a set T of partition tokens and replace occurrences of *Value* by *PartitionedValue* in the abstract domain, as shown in Figure 5. A *partitioned value* is a partial map from partition tokens to ordinary values. We use the notation $[t_1 \mapsto v_1, \ldots, t_k \mapsto v_k]$ (or set-builder notation like $[t_i \mapsto v_i \mid i = 1, \ldots, k]$) to denote the partitioned value that maps t_i to v_i for each $i = 1, \ldots, k$ and is undefined for all other partition tokens.

The partition tokens play a similar role as in trace partitioning [23], but at the level of abstract values. (We explain the differences between value partitioning and traditional trace partitioning in more detail in Section 7.) A partition token intuitively represents a set of execution traces. The special token ANY represents all traces, so the partitioned value $[\text{ANY} \mapsto v]$ has the same meaning as the ordinary value v in the original abstract domain. As an invariant, all partitioned values we use are defined for the token ANY.[6] We extend partitioned values to be total functions $pv\colon T \to \mathit{Value}$ by defining $pv(t) = pv(\text{ANY})$ when $t \notin \mathrm{dom}(pv)$.[7]

Assume $X \in \mathit{AnalysisLattice}$ is the result of analyzing a given program, $\sigma = X(\ell)$ is the abstract state at some location ℓ, and $[\ldots, t \mapsto v, \ldots] = \sigma(r)$ is the partitioned value of some register r. The meaning of such a partitioned value is that for any trace that ends at ℓ and is in the set of traces represented by t, the concrete value of r is in the concretization of the abstract value v.

A *covering*[8] at a location ℓ is a set of partition tokens where the union of the sets of traces they represent is the set of all traces that lead to ℓ. This means that if $\sigma(\mathtt{x}) = [\ldots, t_1 \mapsto v_1, ..., t_k \mapsto v_k, \ldots]$ where $\sigma = X(\ell)$ for some program variable **x** at location ℓ where $\{t_1, \ldots, t_k\}$ is a covering, then for every concrete state in $[\![\ell]\!]$, the value of **x** is in the concretization of at least one of the abstract values $v_1, \ldots, v_k$. For the initial abstract state at the program entry, all partitioned values use the trivial covering $\{\text{ANY}\}$.

Now that we have generalized the abstract domain, it is easy to adjust all transfer functions for the different kinds of nodes to operate on partitioned values instead of ordinary values. As an example, the original transfer function for $r_1 \leftarrow r_2 \oplus r_3$ updates a given abstract state σ by $\sigma(r_1) := \sigma(r_2) \oplus \sigma(r_3)$ (where $\oplus$ applied to abstract values works as in constant propagation).[9]

[6] When we define a partitioned value $[t_i \mapsto v_i \mid i = 1, \ldots, k]$ without an ANY token, an ANY partition is implicitly created with value $v_1 \sqcup \cdots \sqcup v_k$.

[7] In trace partitioning terminology, this use of ANY corresponds to a simple pre-ordering of partition tokens.

[8] For formal definitions of the notions of traces and coverings, see Rival and Mauborgne [23]. Basing our approach on partitions instead of coverings (a *partition* is a covering where all the trace sets are disjoint) could improve precision but would complicate the analysis without much practical benefit.

[9] The actual TAJS analysis also models implicit type conversions.

$$
\begin{array}{rcll}
T & ::= & \textsc{any} & \text{(Section 4)} \\
& | & \textsc{val}\langle N, R, \mathit{Value} \rangle & \text{(Section 5.1)} \\
& | & \textsc{fun}\langle F, C, T \rangle & \text{(Section 5.2)} \\
& | & \textsc{type}\langle N, R, \mathit{Types} \rangle & \text{(Section 5.3)} \\
\\
\mathit{Types} & ::= & \mathsf{undefined} \mid \mathsf{null} \mid \mathsf{number} \mid \mathsf{string} \mid \mathsf{boolean} & \\
& | & \mathsf{object} \mid \mathsf{array} \mid \mathsf{function} \mid \mathsf{regexp} &
\end{array}
$$

Figure 6 Partition tokens used by property name partitioning, free variable partitioning, and type partitioning.

When switching to the domain with partitioned values, we simply replace $\sigma(r_2) \oplus \sigma(r_3)$ by $[t \mapsto pv_2(t) \oplus pv_3(t) \mid t \in \mathrm{dom}(pv_2) \cup \mathrm{dom}(pv_3)$ where $pv_2 = \sigma(r_2)$ and $pv_3 = \sigma(r_3)]$. The other transfer functions and least-upper-bound are adapted similarly.

A small example can illustrate how partitioning can make the analysis relational. Assume the binary operation is equality, $r_1 \leftarrow r_2 == r_3$, and that we have two partitions, t_1 and t_2, where both registers r_2 and r_3 have the value 42 in partition t_1, and both have the value `"foo"` in partition t_2. With partitioning, the value of r_1 becomes $[t_1 \mapsto \texttt{true}, t_2 \mapsto \texttt{true}]$ (i.e., definitely true), whereas without partitioning, r_2 and r_3 both have the value `42|"foo"`, so the value of r_1 becomes AnyBool (i.e., true or false).

To get any advantage of the new abstract domain, we of course need to modify specific transfer functions to selectively introduce partition tokens and further exploit the extra information available regarding relational properties between values. We show how that can be accomplished in Section 5. Those mechanisms rely on some general operations for manipulating the partitions in partitioned values. Most importantly, we use an operation $\uplus$ when introducing new coverings: $pv_1 \uplus pv_2$ where $pv_1, pv_2 \in \mathit{PartitionedValue}$ denotes the combined partitioned value. For each token that is only present in one of pv_1 or pv_2, the new value will be the value for that token, and for each token shared by pv_1 and pv_2, the new value will be the join of the two respective values.

5 Three Instantiations of Value Partitioning

We now present three instantiations of the value partitioning framework. Each of them targets a category of relational properties that are relevant to analysis of JavaScript libraries. Each instantiation introduces a family of partition tokens, as shown in Figure 6, along with some modification of the analysis transfer functions. Each partition token represents a set of traces, as explained in the following.

5.1 Property Name Partitioning

The first use of value partitioning is for improving precision at correlated object property read/write operations as in the motivating example.

Partition tokens for property name partitioning

We introduce a family of partition tokens, $\textsc{val}\langle n, r, v \rangle$, where $n \in N$, $r \in R$, and $v \in \mathit{Value}$. Such a token represents the set of traces where at the last occurrence of n, the value of register r is v. In all $\textsc{val}\langle n, r, v \rangle$ tokens we use in property name partitioning, the node n

$$\sigma(r_3) \;:=\; \begin{cases} \sigma(r_3) \;\uplus\; [\textsc{val}\langle n, r_3, p\rangle \mapsto p \mid p \in \textsc{propNames}(\sigma(r_2))] & \\ \qquad\quad \uplus\; [\textsc{val}\langle n, r_3, \textsc{other}\rangle \mapsto \mathsf{AnyString}] & \text{if } \sigma(r_3)(\textsc{any}) = \mathsf{AnyString} \\ \sigma(r_3) & \text{otherwise} \end{cases}$$

$$\sigma(r_1) \;:=\; \begin{cases} [\textsc{val}\langle n, r_3, p\rangle \mapsto \textsc{readProp}(\sigma(r_2)(\textsc{any}), p) \mid p \in \textsc{propNames}(\sigma(r_2))] & \\ \qquad\quad \uplus\; [\textsc{val}\langle n, r_3, \textsc{other}\rangle \mapsto \texttt{undefined}] & \text{if } \sigma(r_3)(\textsc{any}) = \mathsf{AnyString} \\ [\textsc{any} \mapsto \textsc{readProp}(\sigma(r_2)(\textsc{any}), \sigma(r_3)(\textsc{any}))] & \text{otherwise} \end{cases}$$

Figure 7 Introduction of partitioned values at a dynamic property read node n of the form $r_1 \leftarrow r_2[r_3]$.

is a property read node (i.e., of the form $r_1 \leftarrow r_2[r_3]$), the register r is the one holding the property name in that instruction (i.e., r_3 in $r_1 \leftarrow r_2[r_3]$), and the value v is a property name (i.e., an element of P).[10]

As an example, assume the code from Figure 4 appears inside a loop, and consider the following two traces that both end at n_6:

$$\tau_a = \cdots (n_1, \gamma_{1a})(n_2, \gamma_{2a})(n_3, \gamma_{3a})(n_4, \gamma_{4a})\boldsymbol{(n_5, \gamma_{5a})}(n_6, \gamma_{6a})$$

and

$$\tau_b = \cdots (n_1, \gamma_{1b})(n_2, \gamma_{2b})(n_3, \gamma_{3b})(n_4, \gamma_{4b})\boldsymbol{(n_5, \gamma_{5b})}(n_6, \gamma_{6b})$$

where each "$\cdots$" is a trace prefix leading from the program entry point to this part of the code, $\gamma_{1a}, \ldots, \gamma_{6b}$ are concrete states, and τ_a is a prefix of τ_b. The last occurrence of n_5 (which is the instruction $r_5 \leftarrow r_3[r_4]$) is emphasized in each of the traces. Also assume that the value of the register r_4 is `"foo"` in γ_{5a} and `"bar"` in γ_{5b}. Note that r_4 is the register holding the property name at the n_5 instruction. In this situation, the token $\textsc{val}\langle n_5, r_4, \texttt{"foo"}\rangle$ represents τ_a but not τ_b.

Dynamic property reads

Figure 7 shows the modified transfer function for read-property nodes, $r_1 \leftarrow r_2[r_3]$. The function $\textsc{readProp}(v_1, v_2)$ looks up the abstract value of properties named v_2 in the abstract objects pointed to by v_1 in the current state σ.[11] Property name partitioning is triggered if the property name is not precise (here modeled as $\mathsf{AnyString}$), so in that case we partition the property name r_3 with respect to the properties that appear in the abstract objects pointed to by r_2 (expressed as $\textsc{propNames}(\sigma(r_2))$), and perform the property read for each partition to obtain a partitioned value for r_1. We use the artificial abstract value $\textsc{other} \in \mathit{Value}$ to represent all other properties; for that partition, the result value becomes `undefined`.[12] If

[10] In JavaScript, property names are either strings, which we model in the sub-lattice *Prim*, or symbols, which can be modeled as special heap locations.

[11] Reading an object property is a nontrivial operation in JavaScript because of prototypes, getters, and implicit type conversions. Importantly, the value partitioning mechanism is orthogonal to such JavaScript technicalities.

[12] In our implementation we use a more precise string lattice, which allows us to express more precisely that $\sigma(r_3)$ for the $\textsc{val}\langle n, r_3, \textsc{other}\rangle$ partition is $\mathsf{AnyString} \setminus \textsc{propNames}(\sigma(r_2))$, i.e., any string except for the property names that are covered by other partitions. See also footnote 3.

```
for each $t \in \textsc{chooseCommonCovering}(\sigma(r_2), \sigma(r_3))$:
    $\textsc{writeProp}\big(\sigma(r_1)(\textsc{any}), \sigma(r_2)(t), \sigma(r_3)(t)\big)$
```

Figure 8 Exploiting partitioned values at a dynamic property write node, $r_1[r_2] \leftarrow r_3$.

the property name r_3 is already precise (corresponding to the "otherwise" cases), there is no need to introduce new partitions, so in that case r_3 is unmodified and the result value r_1 is obtained directly using READPROP and the ANY partition token.

Recall that a $\textsc{val}\langle n, r, p\rangle$ token represents the set of traces where at the last occurrence of n, the value of register r is v. To respect this property we need to remove all existing $\textsc{val}\langle n, \_, \_\rangle$ tokens from the abstract state before applying the transfer function for dynamic property reads. (This is safe because every abstract value still has other coverings, in particular $\{\textsc{any}\}$.)

Notice that for both r_3 and r_1, the new partitions use the partition tokens $\textsc{val}\langle n, r_3, p\rangle$ where n is the read-property node. Evidently, the new partition tokens form a covering. Also, this new transfer function respects the interpretation of the newly added $\textsc{val}\langle n, r, p\rangle$ tokens, and due to the partitioning, the resulting abstract states maintain the relation between the involved object property names and values.

Dynamic property writes

Next, we modify the transfer function for dynamic property writes, $r_1[r_2] \leftarrow r_3$, as shown in Figure 8, to take advantage of the partitionings introduced at dynamic property reads. The function $\textsc{writeProp}(v_1, v_2, v_3)$ writes v_3 to the properties named v_2 in the objects referred to by v_1.[13] The function CHOOSECOMMONCOVERING finds a covering shared by the property name $\sigma(r_2)$ and the value to be written $\sigma(r_3)$. (An example is given below.) If multiple such coverings exist, a largest one (i.e., one with the largest number of partition tokens) is selected.[14] Recall that the two values always share the $\{\textsc{any}\}$ covering, which will be used if no other covering exist. When a covering has been chosen, the value is written to the appropriate object property for each partition, thereby exploiting the relational information. In case the $\{\textsc{any}\}$ covering is chosen, the transfer function behaves as the original version without value partitioning.

Example

To better understand property name partitioning, we now explain the mechanism in more detail on the example given in Figure 4. Let us assume that $\sigma(\texttt{p}) = [\textsc{any} \mapsto \mathsf{AnyString}]$, $\sigma(\texttt{x}) = [\textsc{any} \mapsto \mathit{obj}_2]$ and $\sigma(\texttt{y}) = [\textsc{any} \mapsto \mathit{obj}_1]$ where in state σ, obj_1 is the location of an empty abstract object and obj_2 is the location of an abstract object with two properties, `{foo: 1, bar: 2}`. This means when analyzing the read property node $r_5 \leftarrow r_3[r_4]$ we have $\sigma(r_3) = [\textsc{any} \mapsto \mathit{obj}_2]$ and $\sigma(r_4) = [\textsc{any} \mapsto \mathsf{AnyString}]$. Since the property name r_4 is

[13] We omit the details of how the implementation of WRITEPROP in TAJS handles strong/weak updates, setters, and implicit type conversions. Importantly, the value partitioning mechanism is orthogonal to such JavaScript technicalities.

[14] Multiple coverings can arise if, for example, the same property name is used at two different property read operations. We choose the largest covering based on the heuristic that fine-grained coverings lead to higher precision than coarse-grained coverings. The most important consequence of this heuristic is that we avoid the $\{\textsc{any}\}$ covering if others are available. In case of multiple largest ones, an arbitrary one is selected among them.

$$\begin{aligned}
pv \in \mathit{PartitionedValue} &= T \hookrightarrow \mathit{FPValue}\\
fv \in \mathit{FPValue} &= \mathit{FunctionPartitions} \times \mathit{Value}\\
fp \in \mathit{FunctionPartitions} &= \mathcal{P}(T)\\
\mathit{AnalysisLattice} &= C' \times N \to \mathit{State}\\
c \in C' &= \mathit{FunctionPartitions} \times C
\end{aligned}$$

Figure 9 Extensions of the abstract domain for free variable partitioning.

imprecise, the first case in each definition in Figure 7 applies, meaning that value partitioning is triggered. Since $\textsc{propNames}(\sigma(r_2)) = \{$`"foo"`, `"bar"`$\}$, we update r_4 such that $\sigma(r_4)$ equals $[\textsc{val}\langle n, r_4,$ `"foo"`$\rangle \mapsto$ `"foo"`, $\textsc{val}\langle n, r_4,$ `"bar"`$\rangle \mapsto$ `"bar"`, $\textsc{val}\langle n, r_4, \textsc{other}\rangle \mapsto$ AnyString$]$, where n is the read property node. Recall from Section 3 that the operation $\sigma(r_4) := \ldots$ does not only modify r_4 but also the must-equals variables and registers, meaning that this partitioned value is also written to r_2 and `p`. The value being read gets the same partitions, such that $\sigma(r_5)$ becomes $[\textsc{val}\langle n, r_4,$ `"foo"`$\rangle \mapsto 1, \textsc{val}\langle n, r_4,$ `"bar"`$\rangle \mapsto 2, \textsc{val}\langle n, r_4, \textsc{other}\rangle \mapsto$ `undefined`$]$.

When reaching the property write operation $r_1[r_2] \leftarrow r_5$, the state σ contains $\sigma(r_2) = [\textsc{val}\langle n, r_4,$ `"foo"`$\rangle \mapsto$ `"foo"`, $\textsc{val}\langle n, r_4,$ `"bar"`$\rangle \mapsto$ `"bar"`, $\textsc{val}\langle n, r_4, \textsc{other}\rangle \mapsto$ AnyString$]$ and $\sigma(r_5) = [\textsc{val}\langle n, r_4,$ `"foo"`$\rangle \mapsto 1, \textsc{val}\langle n, r_4,$ `"bar"`$\rangle \mapsto 2, \textsc{val}\langle n, r_4, \textsc{other}\rangle \mapsto$ `undefined`$]$. We now apply the transfer function from Figure 8. The two values $\sigma(r_2)$ and $\sigma(r_5)$ share two coverings: $\{\textsc{any}\}$ and $\{\textsc{val}\langle n, r_4,$ `"foo"`$\rangle, \textsc{val}\langle n, r_4,$ `"bar"`$\rangle, \textsc{val}\langle n, r_4, \textsc{other}\rangle\}$. Since the second covering is largest, that one is picked by $\textsc{chooseCommonCovering}(\sigma(r_2), \sigma(r_5))$. We therefore perform three writes corresponding to the abstract assignments obj_1`["foo"]=1`, obj_1`["bar"]=2`, and obj_1`[AnyString]=undefined`; notably, the properties `foo` and `bar` are not mixed together.

5.2 Free Variable Partitioning

We now explain how to leverage value partitioning to gain precision for free variables, such as `func` in line 9 in the motivating example from Figure 1.

Extending the abstract domain

The first step is to extend the abstract domain as shown in Figure 9. The *Value* component in *PartitionedValue* is replaced by *FPValue*, which is a product of *FunctionPartitions* and *Value*. The component *FunctionPartitions* is a set of partition tokens, which we use for tracking which partitions the functions described in the *Value* component may have been declared in. (For instance, for the motivating example from Figure 1, the function declared in lines 7–10 was created in the partitions t'_1, t'_2, and t'_3 so the corresponding abstract values become[15] $(\{t'_1\}, \ell)$, $(\{t'_2\}, \ell)$, and $(\{t'_3\}, \ell)$, where ℓ denotes the location for the created closure.) To preserve this information when analyzing calls to such functions, we also augment the set of contexts to include this information (replacing C by C' in *AnalysisLattice*). The *FunctionPartitions* set is empty for values and contexts that do not use free variable partitioning.

[15] These three abstract values are denoted $\ell_{t'_1}$, $\ell_{t'_2}$, and $\ell_{t'_3}$, respectively, in the motivating example in Section 2.

$$pv(t) = \begin{cases} pv(t) & \text{if } t \in \textsc{dom}(pv) \\ \bot & \text{otherwise if } t = \textsc{fun}\langle f, c, t' \rangle \wedge \\ & \qquad\qquad \exists c', t'' \colon c \neq c' \wedge \textsc{fun}\langle f, c', t'' \rangle \in \textsc{dom}(pv) \\ pv(\textsc{any}) & \text{otherwise} \end{cases}$$

Figure 10 Redefining how partitioned values are extended to total functions, exploiting free variable partitioning.

Next, we introduce a new kind of partition tokens, and we then describe how elements of *FunctionPartitions* are created at function expressions and used at read variable nodes.

Partition tokens for free variable partitioning

We introduce a new kind of partition tokens, $\textsc{fun}\langle f, c, t\rangle$, where f is a function, $c \in C'$ is a context, and $t \in T$ is a partition token. A trace is represented by such a token if (1) the trace ends at a program location that belongs to a closure that was created when the trace up to that point was a t trace, and (2) that point in the trace is in function f in context c. (For instance, in the motivating example, a trace ending in line 9 where the currently executed closure was created in line 7 at the end of a t_1 trace can be represented by $\textsc{fun}\langle f, c, t_1\rangle$, where f is the function at lines 2–11 and c is the context for the call to that function.) We only allow such partition tokens to appear in abstract values of variables that are declared in f. Intuitively, we use these partition tokens to obtain a form of heap specialization (also called heap cloning or context sensitive heap) [20] for the activation objects of f.[16]

An important property is that if the abstract value of a variable `x` declared in a function f contains partition token $\textsc{fun}\langle f, c', t''\rangle$ for some c', t'' but not $\textsc{fun}\langle f, c, t'\rangle$ for any c, t' where $c \neq c'$, then f has not been invoked with context c in any trace represented by $\textsc{fun}\langle f, c, t'\rangle$. This means that it is safe to redefine how partitioned values are extended to total functions as shown in Figure 10. The only difference between the new and the original definition from Section 4 is the second case, where $\bot$ is returned to indicate that the set of traces for the given partition is empty due to the above mentioned property being satisfied.

Function definitions

Assume the analysis reaches a function definition node, $r_1 \leftarrow \mathsf{function}(\cdots)\{\cdots\}$, while analyzing a function f in context c, and that the function being defined has free variables $\mathtt{x}_1, \ldots, \mathtt{x}_n$ that are declared in f (i.e., as parameters or local variables). Note that f is the function containing the function definition node being analyzed, not the function being defined. Let ℓ denote the location of the newly created closure according to the original transfer function without free variable partitioning. We now partition both the resulting function value of register r_1 and the values of $\mathtt{x}_1, \ldots, \mathtt{x}_n$ as shown in Figure 11.

First, we use a function $\textsc{chooseCovering}$ that finds a largest covering, denoted LC, among the values of $\mathtt{x}_1, \ldots, \mathtt{x}_n$. (If multiple such coverings exist, an arbitrary one is selected among them, as before.)

[16] Local variables and arguments are stored as properties on activation objects, which are created on each invocation.

$$LC = \textsc{chooseCovering}(\sigma(\mathtt{x}_1), \ldots, \sigma(\mathtt{x}_n))$$

$$\sigma(\mathtt{x}_i) \ := \ \begin{cases} \sigma(\mathtt{x}_i) \uplus [\textsc{fun}\langle f,c,t\rangle \mapsto \sigma(\mathtt{x}_i)(t) \mid t \in LC] & \text{if } LC \subseteq \mathrm{dom}(\sigma(\mathtt{x}_i)) \\ \sigma(\mathtt{x}_i) & \text{otherwise} \end{cases}$$

$$\sigma(r_1) \ := \ \begin{cases} [t \mapsto (\{\textsc{fun}\langle f,c,t\rangle\} \cup \mathit{fp}, \ell) \mid t \in LC] & \text{if } LC \subseteq \mathrm{dom}(\sigma(\mathtt{x}_i)) \text{ for some } \mathtt{x}_i \\ [\textsc{any} \mapsto (\emptyset, \ell)] & \text{otherwise} \end{cases}$$

Figure 11 Introduction of partitioned values at a function definition node $r_1 \leftarrow \mathsf{function}(\cdots)\{\cdots\}$.

For each $\mathtt{x}_i$ for $i = 1, \ldots, n$, if the current value of $\mathtt{x}_i$ contains the covering LC, we add $\textsc{fun}\langle f,c,t\rangle \mapsto \sigma(\mathtt{x}_i)(t)$ to the value of $\mathtt{x}_i$ for each $t \in LC$. (This evidently respects the meaning of $\textsc{fun}\langle f,c,t\rangle$ tokens informally described in the beginning of the section.) Otherwise, if the current value of $\mathtt{x}_i$ does not contain LC, we leave $\mathtt{x}_i$ unmodified.

For the result register r_1, we augment the function location ℓ by the same partition tokens. If at least one free variable has been partitioned (i.e., $LC \subseteq \mathrm{dom}(\sigma(\mathtt{x}_i))$ for some $\mathtt{x}_i$), then for each of the partition tokens $t \in LC$, the value of r_1 becomes the augmented value $(\{\textsc{fun}\langle f,c,t\rangle\} \cup \mathit{fp}, \ell)$ where fp is the set of function partitions in the current context c. By augmenting the value using the $\textsc{fun}\langle f,c,t\rangle$ token, the information about the partitioning is available when ℓ is later invoked, which is explained below. (The function partitions fp of the current context describe how the current function was declared in an outer scope, so by inheriting those, the partitioning also works for multiple layers of nested functions.) Otherwise, if none of the free variables have been partitioned, register r_1 is assigned the partitioned value $[\textsc{any} \mapsto (\emptyset, \ell)]$, which is equivalent to the original transfer function without free variable partitioning.

Function calls

At a function call $r_1 \leftarrow r_2(r_3)$ where $\sigma(r_2)$ is an augmented function value (fp, v) (i.e., fp is a set of partition tokens introduced at function definitions and v refers to the set of closures that may be invoked), we use fp to augment the context for each callee. (The set of augmented contexts C' contains the *FunctionPartitions* component exactly for this purpose.) Assume for simplicity that v refers to a single closure location so we only have one callee. By augmenting the context, when analyzing the body of the callee we retain the information about the partitions where the callee closure was created, which we can exploit when reading its free variables as explained next.

Variable reads

Figure 12 shows the updated transfer function for read variable nodes $r_1 \leftarrow \mathtt{x}$, where we read a variable $\mathtt{x}$ in a calling context with function partitions fp. The set of function partitions fp tells us which partitions the current closure may have been created in. For this reason, if the abstract value of $\mathtt{x}$ contains partition tokens that are also in fp, we can obtain a covering for $\mathtt{x}$ by considering only those partition tokens. If there is no such partition token, we just read the value of $\mathtt{x}$ as in the original transfer function.

$$\sigma(r_1) \;:=\; \begin{cases} [\textsc{any} \mapsto \bigsqcup\{\sigma(\mathtt{x})(t) \mid t \in \mathrm{dom}(\sigma(\mathtt{x})) \cap \mathit{fp}\}] & \text{if } \mathrm{dom}(\sigma(\mathtt{x})) \cap \mathit{fp} \neq \emptyset \\ \sigma(\mathtt{x}) & \text{otherwise} \end{cases}$$

Figure 12 Exploiting partitioned values at a read variable node, $r_1 \leftarrow \mathtt{x}$.

```
22 function f(v) {
23   return function g() {
24     return v;
25   }
26 }
27
28 var foo = f("foo");
29 var bar = f("bar");
30
31 assert(bar() != "foo");
```

Figure 13 Free variable partitioning example with different contexts.

```
32 var o1 = {x: 1, y: 2};
33 var o2 = {};
34 Object.keys(o1).forEach(
35   function h(p) {
36     var v = o1[p];
37     o2[p] = function j() {
38       return v;
39     }
40   }
41 );
42 assert(o2.y() != 1);
```

Figure 14 Free variable partitioning example with partitioned argument.

As an example, assume $\sigma(\mathtt{x}) = [\textsc{fun}\langle f, c, t\rangle \mapsto 1, \textsc{fun}\langle f, c', t'\rangle \mapsto 2, \ldots]$ and $\mathit{fp} = \{\textsc{fun}\langle f, c, t\rangle\}$. The value of `x` tells us that `x` must be a local variable in function f which may have been called in contexts c and context c', and that `x`'s value is 1 or 2, respectively. Since $\mathit{fp} = \{\textsc{fun}\langle f, c, t\rangle\}$, we know that the current function is defined inside the lexical scope of f in context c, meaning that the value of `x` must be 1.

Examples

To better understand free variable partitioning, we provide two examples. The first example (Figure 13) shows how free variable partitioning can preserve precision when a function is called in multiple contexts, in a way that resembles traditional heap specialization [20]. The second example (Figure 14) shows how free variable partitioning can preserve the precision of free variables partitioned with property name partitioning.

In Figure 13, lines 22–26 define a function `f` that returns a closure, which on invocation returns the argument passed to `f`. Lines 28 and 29 call `f` with the arguments `"foo"` and `"bar"` and store the returned closures in the variables `foo` and `bar`, respectively. Line 31 calls the closure stored in `bar` and asserts that the resulting value is not the string `"foo"`. The two calls to `f` are analyzed in different contexts c and c' (due to the context sensitivity mechanism mentioned in Section 3, as `"foo"` and `"bar"` are determinate values). For the invocation `bar()`, the resulting value is the value of the free variable `v` in the closure stored in `bar`. If not using heap specialization, the two concrete activation objects at the two calls to `f` would be modeled by a single abstract object, so the free variable `v` would have the imprecise abstract value AnyString. To reason precisely about the assertion in line 31, the analysis has to distinguish the value of `v` at the two calls. The baseline TAJS analyzer accomplishes this by the use of heap specialization [2], which provides two different abstract activation objects for the calls to `f`, so the two values `"foo"` and `"bar"` are kept separate.

With free variable partitioning we obtain the same degree of precision as with heap specialization in this situation. Since `v` is a free variable in the closure created in line 23, we apply the top cases in the transfer functions shown in Figure 11 with $LC = \{\textsc{any}\}$. This means that `v` after the call to `f("foo")` will have the value $[\textsc{any} \mapsto$ `"foo"`$, \textsc{fun}\langle f, c, \textsc{any}\rangle \mapsto$ `"foo"`$]$ and the value written to the `foo` variable is $(\{\textsc{fun}\langle f, c, \textsc{any}\rangle\}, \ell_g)$ where ℓ_g is the location of the closure created in line 23. For the call to `f("bar")`, the value for `v` will similarly be $[\textsc{any} \mapsto$ `"bar"`$, \textsc{fun}\langle f, c', \textsc{any}\rangle \mapsto$ `"bar"`$]$ and the value written to `bar` is $(\{\textsc{fun}\langle f, c', \textsc{any}\rangle\}, \ell_g)$. Note the difference in the context part of the $\textsc{fun}$ token (c at the `"foo"` call and c' at the `"bar"` call), since the calls to `f` are in those two different contexts. The value of `v` then becomes $[\textsc{any} \mapsto \mathsf{AnyString}, \textsc{fun}\langle f, c, \textsc{any}\rangle \mapsto$ `"foo"`$, \textsc{fun}\langle f, c', \textsc{any}\rangle \mapsto$ `"bar"`$]$, so that the $\textsc{fun}$ partitions preserve the precise values.

Now when analyzing `bar()`, `bar` has the value $(\{\textsc{fun}\langle f, c', \textsc{any}\rangle\}, \ell_g)$, which means the calling context to the function `g` is augmented with the set of function partitions $\{\textsc{fun}\langle f, c', \textsc{any}\rangle\}$ as described above. When reading the free variable `v` in line 24, we use the first case in the transfer function defined in Figure 12, since $\mathrm{dom}(\sigma(\mathtt{x})) \cap \mathit{fp}$ is $\{\textsc{fun}\langle f, c', \textsc{any}\rangle\}$. This means that the resulting value from the variable read is the value $[\textsc{any} \mapsto$ `"bar"`$]$, so we obtain the same precision as with heap specialization.

This first example shows how the free variable partitioning mechanism works and how it relates to heap specialization, but it does not demonstrate any precision improvements compared to the existing TAJS analyzer, which does apply heap specialization. The second example, Figure 14, illustrates a simplified version of how free variable partitioning was used in the motivating example in combination with property name partitioning, which leads to a precision improvement of TAJS. Line 32 defines the object `o1` with two properties, and line 33 defines `o2` as an empty object. Lines 34–41 iterate over the properties of `o1`. For each property, it writes a function returning the value of `o1[p]` to the `p` property of `o2`. To prove that the assertion at line 42 always holds, it is critical that the values of `v` are not mixed together in the iterations.

Using property name partitioning at line 36, the value of `v` becomes $[\textsc{val}\langle n, r, \texttt{"x"}\rangle \mapsto 1, \textsc{val}\langle n, r, \texttt{"y"}\rangle \mapsto 2]$ and the value of `p` becomes $[\textsc{val}\langle n, r, \texttt{"x"}\rangle \mapsto \texttt{"x"}, \textsc{val}\langle n, r, \texttt{"y"}\rangle \mapsto \texttt{"y"}]$, where n is the read property node and r is the register storing the property name. (For clarity we ignore the $\textsc{other}$ partition in this example.) When analyzing the closure creation at line 37, we use the top rules in Figure 11 with $LC = \{\textsc{val}\langle n, r, \texttt{"x"}\rangle, \textsc{val}\langle n, r, \texttt{"y"}\rangle\}$. This means that `v` is augmented with the additional partitions $[\textsc{fun}\langle \mathtt{h}, c, \textsc{val}\langle n, r, \texttt{"x"}\rangle\rangle \mapsto 1, \textsc{fun}\langle \mathtt{h}, c, \textsc{val}\langle n, r, \texttt{"y"}\rangle\rangle \mapsto 2]$, and the value being written to `o2[p]` is $[\textsc{val}\langle n, r, \texttt{"x"}\rangle \mapsto (\{\textsc{fun}\langle \mathtt{h}, c, \textsc{val}\langle n, r, \texttt{"x"}\rangle\rangle\}, \ell_j), \textsc{val}\langle n, r, \texttt{"y"}\rangle \mapsto (\{\textsc{fun}\langle \mathtt{h}, c, \textsc{val}\langle n, r, \texttt{"y"}\rangle\rangle\}, \ell_j)]$. Here, ℓ_j denotes the location of the closure created in line 37. At the dynamic property write, the property name and value to be written share the covering $\{\textsc{val}\langle n, r, \texttt{"x"}\rangle, \textsc{val}\langle n, r, \texttt{"y"}\rangle\}$, meaning that the write happens as described in Figure 8, so that `o2.x` becomes $(\{\textsc{fun}\langle \mathtt{h}, c, \textsc{val}\langle n, r, \texttt{"x"}\rangle\rangle\}, \ell_j)$ and `o2.y` becomes $(\{\textsc{fun}\langle \mathtt{h}, c, \textsc{val}\langle n, r, \texttt{"y"}\rangle\rangle\}, \ell_j)$. Now when `o2.y` is called in line 42, the call to `j` is augmented with the the set of function partitions $\{\textsc{fun}\langle \mathtt{h}, c, \textsc{val}\langle n, r, \texttt{"y"}\rangle\rangle\}$. Therefore when reading the value `v` in line 38, according to Figure 12 we only read the $\textsc{fun}\langle \mathtt{h}, c, \textsc{val}\langle n, r, \texttt{"y"}\rangle\rangle$ partition. The result of reading `v` is then $[\textsc{any} \mapsto 2]$, so the analysis is precise enough to prove that the assertion at line 42 holds.

5.3 Type Partitioning

Value partitioning can also be useful for partitioning values based on their types. Since JavaScript does not have function overloading, it is common to reflectively find the type of an argument, and based on the type run different pieces of code (as in line 3 in Figure 1).

$$\sigma(r_3) \; := \; \begin{cases} \sigma(r_3) \uplus [\textsc{type}\langle n, r_3, ty\rangle \mapsto \textsc{filter}(\sigma(r_3), ty) \mid ty \in \textsc{types}(\sigma(r_3))] \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{if } |\textsc{types}(\sigma(r_3))| > 1 \\ \sigma(r_3) \qquad\qquad\qquad\qquad\qquad\qquad\qquad\;\; \text{otherwise} \end{cases}$$

Figure 15 Addition to the transfer function for a call node with one argument, $r_1 \leftarrow r_2(r_3)$. FILTER restricts a partitioned value to represent only values that match the given type, and TYPES returns the possible types of the given partitioned value.

This is often done through the use of predicate functions, which are one-parameter functions that return a boolean value. By partitioning the arguments at calls to predicate functions, the analysis becomes able to track the relations between the arguments and the return values, and thereby boost the control sensitivity mechanism (see Section 3) at branches that involve such calls. Since the analysis does not know in advance whether a function returns boolean values, we simply perform this partitioning at all function calls with one argument, without considering what values the function may return.

Partition tokens for type partitioning

We introduce type partitioning tokens of the form $\textsc{type}\langle n, r, ty\rangle$, where $n \in N$ is a call node $r_1 \leftarrow r_2(r_3)$, $r \in R$ is the argument register in n (in this case r_3), and $ty \in \mathit{Types}$ using the set of types shown in Figure 6. Such a token represents the set of traces where the type of r is ty at the last occurrence of n. For example, the traces that reach line 7 in Figure 1 are represented by the token $\textsc{type}\langle n, r, \mathsf{function}\rangle$ where n is the call to `isFunction` in line 6 and r is the argument register of that call node.

Function calls

Figure 15 shows an addition to the transfer function for call nodes, $r_1 \leftarrow r_2(r_3)$, to partition the argument value before the call takes place. The first case applies if the argument $\sigma(r_3)$ abstractly represents values of multiple types (i.e., $|\textsc{types}(\sigma(r_3))| > 1$, where TYPES returns the set of all the types the given abstract value may have). In this case we introduce a partition $\textsc{type}\langle n, r_3, ty\rangle$ for each $ty \in \textsc{types}(\sigma(r_3))$, such that the value in that partition is $\textsc{filter}(\sigma(r_3), ty)$, where FILTER restricts $\sigma(r_3)$ to only represent values of type ty. Since all the possible types are represented, the new partitions together form a covering.

Recall that a $\textsc{type}\langle n, r, ty\rangle$ token only represents information about the last occurrence of n in a given trace. To ensure this property we always remove all existing $\textsc{type}\langle n, \_, \_\rangle$ tokens from the abstract state immediately before applying the modified transfer function for call node n.

Example

As an example consider the code in Figure 16, and assume `x` has the abstract value `fun1`|`obj2` (representing either the function `fun1` or the object `obj2`). Without type partitioning, the result of analyzing the `isObj(x)` call is the abstract value AnyBool (representing true or false), so both branches are analyzed with `x` being `fun1`|`obj2`; however, in a concrete execution, `fun1` will never flow to the "true" branch, and `obj2` will never flow to the "false" branch.

```
function isObj(arg) {
  return typeof arg == 'object';
}
if (isObj(x)) { ... } else { ... }
```

Figure 16 Type partitioning example.

```
47 function isObj(arg) {
48   if (typeof arg == 'object')
49     return true;
50   else
51     return false;
52 }
53 if (isObj(x)) { ... } else { ... }
```

Figure 17 Type partitioning example with control dependent relations.

By using type partitioning, we partition `x` before calling the predicate function. In this example let n be the call node and let r be its argument register. Then `x` becomes $[\textsc{type}\langle n, r, \mathsf{function}\rangle \mapsto \texttt{fun1}, \textsc{type}\langle n, r, \mathsf{object}\rangle \mapsto \texttt{obj2}]$. Now when analyzing the body of `isObj`, the expression `typeof arg == 'object'` evaluates to the partitioned value $[\textsc{type}\langle n, r, \mathsf{function}\rangle \mapsto \texttt{false}, \textsc{type}\langle n, r, \mathsf{object}\rangle \mapsto \texttt{true}]$. When reaching the `if` branch, control sensitivity ensures that only the object partition flows to the "true" branch (i.e., `x`'s value becomes $[\textsc{type}\langle n, r, \mathsf{function}\rangle \mapsto \bot, \textsc{type}\langle n, r, \mathsf{object}\rangle \mapsto \texttt{obj2}]$ in that branch), and only the function partition flows to the "false" branch (i.e., `x`'s value becomes $[\textsc{type}\langle n, r, \mathsf{function}\rangle \mapsto \texttt{fun1}, \textsc{type}\langle n, r, \mathsf{object}\rangle \mapsto \bot]$ in that branch).

Control dependent relations

Predicate functions are sometimes implemented with control dependent relations between the argument and the result, as in the example in Figure 17. The example is contrived but it is not uncommon in predicate functions that the result values appear as the literals `true` or `false` in branches. With the type partitioning mechanism described above, the returned values will not be partitioned in this situation, since the partitions in `arg` do not propagate to the values `true` and `false`.

To mitigate this issue, we augment the abstract states as shown in Figure 18 to keep track of partitions that must be dead or may be live (represented by the two $\mathcal{P}(T)$ components, respectively). A partition is *dead* if the set of traces it represents is empty, and it is live otherwise. (We only keep track of the live partitions in coverings where there are any dead partitions.) Since the branch condition `typeof arg == 'object'` is analyzed with a partitioned value for `arg`, by control sensitivity we know that the only traces that can reach the "true" branch are those represented by the object partition, so we record that $\textsc{type}\langle n, r, \mathsf{object}\rangle$ is live and $\textsc{type}\langle n, r, \mathsf{function}\rangle$ is dead in that branch, and conversely in the other branch. To exploit this information, we also update the transfer function for constants, $r_1 \leftarrow c$, as shown in Figure 19. Basically, it assigns $\bot$ to all dead partitions and the constant c to all live partitions. If there are no dead partitions, it behaves as usual, where the constant is written to the $\textsc{any}$ partition. When the analysis reaches `true` (line 49), we obtain the partitioned value $[\textsc{type}\langle n, r, \mathsf{function}\rangle \mapsto \bot, \textsc{type}\langle n, r, \mathsf{object}\rangle \mapsto \texttt{true}]$, and similarly when analyzing `false` (line 51) we get $[\textsc{type}\langle n, r, \mathsf{function}\rangle \mapsto \texttt{false}, \textsc{type}\langle n, r, \mathsf{object}\rangle \mapsto \bot]$. The join of these two values is $[\textsc{type}\langle n, r, \mathsf{function}\rangle \mapsto \texttt{false}, \textsc{type}\langle n, r, \mathsf{object}\rangle \mapsto \texttt{true}]$, which becomes the result of `isObj(x)`. Due to the control sensitivity mechanism, only `obj2` then flows to the "true" branch, and only `fun1` flows to the "false" branch in line 53.

$$\mathit{State}' = \mathit{State} \times \mathcal{P}(T) \times \mathcal{P}(T)$$

Figure 18 Abstract states updated to keep track of dead and live partitions.

$$\sigma(r_1) := \begin{cases} [t \mapsto c \mid t \in \textsc{livePartitions}(\sigma)] \uplus [t \mapsto \bot \mid t \in \textsc{deadPartitions}(\sigma)] \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{if } \textsc{deadPartitions}(\sigma) \neq \emptyset \\ [\textsc{any} \mapsto c] \qquad\qquad\qquad\qquad\qquad\qquad \text{otherwise} \end{cases}$$

Figure 19 Updated transfer function for constant nodes, $r_1 \leftarrow c$, for improved type partitioning.

6 Evaluation

We have implemented the value partitioning framework (Section 4) and the three instantiations (Section 5) on top of TAJS v0.24. Implementing the general framework in TAJS required 900 lines of code, however most of this is boilerplate code for lifting operations on ordinary abstract values to also work on partitioned values. With the general framework in place, instantiations are easy to implement: property name partitioning (Section 5.1), free variable partitioning (Section 5.2), and type partitioning (Section 5.3) required only around 230, 250, and 60 lines of code, respectively. We disable TAJS's `for-in` specialization technique, since it is subsumed by property name partitioning.[17] We refer to our new analysis tool as TAJS$_{\text{ValPar}}$.[18] Using this tool we evaluate our techniques by answering the following research questions:

RQ1 How does TAJS$_{\text{ValPar}}$ compare to existing state-of-the-art analyses for JavaScript?
RQ2 What are the effects of the three different instantiations of value partitioning?

All our experiments are conducted on an Ubuntu machine with a 2.6 GHz Intel Xeon E5-2697A CPU running a JVM with 10 GB RAM.

6.1 RQ1: Comparison with State-Of-The-Art Analyses

We start by comparing TAJS$_{\text{ValPar}}$ against the current state-of-the-art analyses for JavaScript: the baseline TAJS analyzer with static determinacy [2], TAJS$_{\text{VR}}$ [26] with demand-driven value refinement, and the CompAbs analyzer [16] based on the SAFE analyzer [17]. We use the same benchmarks as those used in the evaluation of TAJS$_{\text{VR}}$, which is the most recent related work.

Micro benchmarks

We first evaluate TAJS$_{\text{ValPar}}$ against a small collection of micro benchmarks that capture some of the main challenges that appear in analysis of modern JavaScript libraries and are used in previous work [16, 26]. The benchmarks all contain dynamic read/write pairs that

[17] The motivation for introducing `for-in` specialization in [2] was to reason about correlated read/write pairs inside `for-in` loops. This relational information is now provided by property name partitioning.
[18] TAJS$_{\text{ValPar}}$: **TAJS** with **Val**ue **Par**titioning

Table 1 Micro-benchmarks that check how state-of-the-art analyses handle various dynamic read/write pairs that represent typical challenges in JavaScript library code. 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 [16], while M1, M2, and M3 are drawn directly from [26].

Benchmark	TAJS	CompAbs	TAJS_{VR}	$\text{TAJS}_{\text{ValPar}}$
CF	✓	✓	✓	✓
CG	✓	✓	✓	✓
AF	✗	✓	✓	✓
AG	✗	✓	✓	✓
M1	✗	✗	✓	✓
M2	✗	✗	✓	✓
M3	✗	✗	✓	✓

are variations of the pattern shown in the introduction and the motivating example. The results of the comparison are shown in Table 1. For these benchmarks, a test *succeeds* if it avoids mixing together properties in the dynamic read/write pairs.

The first two examples, CF and CG, are loops where the static analyses have enough information to be able to unroll all the iterations and thereby analyze the read/write patterns with precise property names. For CF, property name partitioning in $\text{TAJS}_{\text{ValPar}}$ gives the same degree of precision without loop unrolling.

AF and AG are loops where the static analyses are incapable of obtaining a precise value for the property name used in the dynamic read/write pairs. TAJS fails to analyze these, but CompAbs detects the pattern syntactically and therefore applies trace partitioning to analyze the code precisely. TAJS_{VR} also succeeds on these tests, because its backwards abstract interpreter is capable of providing the necessary relational information. In comparison, $\text{TAJS}_{\text{ValPar}}$ can reason about the relational information on its own.

Both TAJS and CompAbs fail on the last three tests (M1, M2, and M3). CompAbs fails on M1 and M3 because it does not apply partitioning due to the fragility of syntactic patterns, and it fails on M2 because the partitioning does not provide the necessary precision about free variables. Again, TAJS_{VR} can analyze them all, since the backwards abstract interpreter is powerful enough to reason about all the cases, whereas $\text{TAJS}_{\text{ValPar}}$ successfully preserves the relational properties by the use of value partitioning.

These results demonstrate that for these benchmarks, $\text{TAJS}_{\text{ValPar}}$ is capable of providing comparable precision to the demand-driven value refinement technique without the need for a complicated backwards analysis, and provides better precision than the other analyses.

Library benchmarks

The next set of benchmarks is taken from the evaluation of TAJS_{VR} and consists of small test cases for popular real-world libraries. The libraries include the widely used functional utility library Underscore (which has more than 20 000 dependents in npm) v1.8.3 with 1 548 LoC and the most depended-upon package Lodash (more than 115 000 dependents). We analyze both Lodash3 (v3.0.0, 10 785 LoC) and Lodash4 (v4.17.10, 17 105 LoC), since their code bases are substantially different and therefore pose distinct challenges for static analysis.

Table 2 Analysis results for real-world benchmarks (from [26]). For each group of benchmarks and for each of the four analyzers, we show the number of tests that are analyzed successfully and (in parentheses) the average analysis time per successful test.

Benchmark group	**TAJS**	**CompAbs**	**TAJS$_{\text{VR}}$**	**TAJS$_{\text{ValPar}}$**
Underscore (182 tests)	0 (-)	0 (-)	173 (2.9s)	173 (2.7s)
Lodash3 (176 tests)	7 (2.4s)	0 (-)	172 (5.5s)	173 (5.3s)
Lodash4 (306 tests)	0 (-)	0 (-)	266 (24.7s)	289 (26.3s)
Prototype (6 tests)	0 (-)	2 (23.1s)	5 (97.7s)	5 (34.1s)
Scriptaculous (1 tests)	0 (-)	1 (62.0s)	1 (236.9s)	1 (55.2s)
jQuery (71 tests)	3 (16.0s)	0 (-)	3 (13.5s)	3 (20.4s)

The other libraries, Prototype v1.7.2, Scriptaculous v1.9.0, and jQuery v1.10,[19] are popular libraries for client-side web programming.

The analysis results are shown in Table 2. We classify an analysis of a benchmark as successful if it terminates within 5 minutes and the analysis result to our knowledge is sound. In particular, an analysis run is considered a failure if the analysis result does not have dataflow to the ordinary exit of the program. (All the tests pass in normal execution, so an analysis result is obviously unsound if there is no dataflow to the ordinary exit.) To increase confidence in the soundness of the analysis results for TAJS$_{\text{ValPar}}$, we apply thorough soundness testing as described at the end of this section. Increasing the time budget does not help for these benchmarks: as reported previously for JavaScript analysis tools, critical precision losses tend to cause a proliferation of spurious dataflow that drastically increases analysis time and renders the analysis results useless [26, 11, 22, 16].

The results for TAJS$_{\text{ValPar}}$ are comparable to those of TAJS$_{\text{VR}}$, which outperforms the other analyzers. TAJS$_{\text{ValPar}}$ succeeds in analyzing all the benchmarks that TAJS$_{\text{VR}}$ can handle, plus 24 more (one Lodash3 test and 23 Lodash4 tests). Note the substantial improvement for the Lodash4 tests: the number of Lodash4 tests that are not analyzed successfully is reduced from 40 to 17. None of the analyzers do well on the jQuery benchmarks; a preliminary manual study shows that the reasons are unrelated to relational analysis. The results are as expected, since property name partitioning and free variable partitioning are alternative techniques to provide the relational information that TAJS$_{\text{VR}}$ obtains from its demand-driven value refinement. Furthermore, value partitioning is triggered more often during the analysis, which means that the precision improvements are not limited to the few critical cases where value refinement is triggered. On top of this, type partitioning provides some additional precision beyond the capabilities of TAJS$_{\text{VR}}$.

Comparing the performance between TAJS$_{\text{ValPar}}$ and TAJS$_{\text{VR}}$, the most significant differences are for the Prototype and Scriptaculous benchmarks. TAJS$_{\text{ValPar}}$ is around 3–4 times faster than TAJS$_{\text{VR}}$, which is mainly because property name partitioning makes the `for-in` specialization technique in TAJS obsolete. For Underscore and Lodash3, TAJS$_{\text{ValPar}}$ is slightly faster than TAJS$_{\text{VR}}$. This is encouraging, because analyzing dynamic property writes as the one in line 7 in Figure 1 is more expensive in TAJS$_{\text{ValPar}}$ than in TAJS$_{\text{VR}}$. In TAJS$_{\text{VR}}$ such an operation is handled as a single imprecise write (since the precision is recovered on demand), whereas TAJS$_{\text{ValPar}}$ performs the write for each property that is copied. To soundly handle setters, all the writes happen in different states that are

[19] This is the version of jQuery used in [2]. Note that [16] used the older v1.4.4.

subsequently joined together, which causes $\text{TAJS}_{\text{ValPar}}$ to spend some extra time at such writes. Since the analysis time is nevertheless similar, we can conclude that value partitioning is cheaper for analyzing other parts of the libraries. For Lodash4 and jQuery, TAJS_{VR} is slightly faster than $\text{TAJS}_{\text{ValPar}}$. For Lodash4, the main reason is the handling of dynamic property writes, and for the jQuery benchmarks, type partitioning adds little performance overhead as seen in Table 3.

Precision

Previous work [2, 22, 26] established that type analysis and call-graph construction are useful metrics for measuring the precision of an analysis for JavaScript, and therefore we use these metrics to evaluate the analysis precision of $\text{TAJS}_{\text{ValPar}}$. All locations are treated context-sensitively in these measurements, meaning that we count the same location once for each reachable context. We count the number of possible types for the resulting value in each variable or property read and find that in 99.19% of the reads, a single unique type is read, with the average number of types being 1.02. For measuring precision of the call-graph construction, we measure the number of call-sites with unique callees, and find this number to be 99.95% of all call-sites. These numbers show that when the analysis succeeds, it does so with very high precision.

Soundness

Formally proving soundness of the three variants of value partitioning is out of scope of this paper, however, we will informally justify that the general approach is sound. Since general trace partitioning is known to be sound, it suffices to argue that the precision gained by value partitioning is equivalent to that obtained through trace partitioning. The key reason why this holds for property name partitioning and type partitioning is that the partition tokens represent the last occurrence of some node, meaning that if two values share partitions, they represent information about the same execution traces. This means that we could (if ignoring performance) instead have applied traditional trace partitioning, with exactly the same partition tokens and at the same nodes, resulting in the same precision. (For further discussion about the connection between value partitioning and trace partitioning, see Section 7.) Similarly for free variable partitioning, since the partitions are only allowed on activation objects, the precision is never higher than what would be obtained using heap specialization (where each partition would be represented by a distinct abstract activation object), and therefore soundness follows from soundness of heap specialization.

Furthermore, to increase confidence in the soundness of our implementation, all the $\text{TAJS}_{\text{ValPar}}$ results have been thoroughly soundness tested [3]. This means that the analysis results overapproximate all the dataflow facts that have been observed during concrete executions of the analyzed benchmarks. For every variable and property read observed concretely, we have checked that the concrete value is in the concretization of the corresponding abstract value in the analysis results, and similarly for property writes and function calls. All our benchmarks except one pass in total more than 7.6 million soundness tests. The one benchmark that fails is a Lodash4 test, which uses ES6 iterators in combination with `Arrays.from`, which is not fully supported in the latest version of TAJS and is unrelated to the use of value partitioning.

Table 3 Analysis results for real-world benchmarks (from [26]) using different instantiations of value partitioning. "None" is without value partitioning, "P" is with property name partitioning, "P + FV" is with property name and free variable partitioning, and "F + PV + T" is with property name, free variable, and type partitioning.

Benchmark group		None		P		P + FV		P + FV + T	
Underscore	(182 tests)	0	(-)	149	(2.0s)	173	(2.5s)	173	(2.7s)
Lodash3	(176 tests)	7	(2.4s)	167	(4.7s)	173	(5.1s)	173	(5.3s)
Lodash4	(306 tests)	0	(-)	268	(16.8s)	274	(27.7s)	289	(26.3s)
Prototype	(6 tests)	0	(-)	0	(-)	5	(32.7s)	5	(34.1s)
Scriptaculous	(1 tests)	0	(-)	0	(-)	1	(53.1s)	1	(55.2s)
jQuery	(71 tests)	3	(16.0s)	3	(15.2s)	3	(16.5s)	3	(20.4s)

6.2 RQ2: Effects of the Three Instantiations

We now investigate how much each of the three uses of value partitioning contributes to the results reported in the previous section. The results from running our analysis with only some instantiations enabled can be seen in Table 3. The column "P" is with only property name partitioning enabled; we see that it is sufficient for analyzing many of the Underscore and Lodash test cases, but not for any of the Prototype or Scriptaculous test cases. (Without property name partitioning but with the other two instantiations enabled, the analysis is not able to analyze more benchmarks than TAJS.) The column "P + FV" uses both property name partitioning and free variable partitioning. Also enabling free variable partitioning makes the analysis capable of analyzing many additional benchmarks: more Underscore and Lodash test cases, as well as some Prototype and Scriptaculous test cases. Compared to only property name partitioning, the analysis times are higher (for the reason discussed above regarding additional state joins). The last column "P + FV + T" is with all instantiations enabled and therefore contains the same numbers as shown in Table 2. We see that type partitioning enables the analysis of 15 additional Lodash4 tests, without significantly increasing the analysis time.

We conclude that all three instantiations contribute to the results, where property name partitioning is the most important one, followed by free variable partitioning and then type partitioning. (TAJS already performs filtering at branches, as mentioned in Section 3; without that feature the effect of type partitioning would likely be larger.)

7 Related Work

Trace partitioning

Value partitioning can be viewed as a variant of trace partitioning [23] as explained in Sections 1, 2 and 4, but there are some important differences. Changing the original abstract domain in Section 4 to support traditional trace partitioning can be done by replacing $L \to \mathit{State}$ by $L \to T \to \mathit{State}$, so that an abstract state is maintained for each partition, at every location. Thus, different locations can partition the abstract states differently. Value partitioning instead has only one abstract state per location but partitions the individual abstract values, which adds an additional degree of flexibility: different parts of each abstract state can be partitioned differently. In particular, for the large parts of the states where we are not interested in relational information, we can use the {ANY} partitioning,[20] while for the important registers and object properties, we can have nontrivial partitions. With traditional trace partitioning, the normal transfer functions are applied for each partition, which causes redundant computations because of the similarities between the different partitions[21]; with value partitioning, we only pay a price for partitioning at operations that involve abstract values with nontrivial partitions. This is the main reason for the low overhead of the technique.

Another difference is that the partition tokens in traditional trace partitioning are actually lists of "directives" (the language of directives used by Rival and Mauborgne [23] is similar to our language of tokens in Figure 6), which can lead to a combinatorial explosion. By partitioning at the level of values and allowing multiple coverings in each partitioned value, we avoid the need to maintain such combinations.

Relational analysis

Traditional techniques for achieving relational analysis, as exemplified by the octagon abstract domain [19], focus on numeric relations, such as, linear inequalities. To reduce the cost of this approach, a syntactic pre-analysis called variable packing is typically used for partitioning the set of program variables, and one octagon is then used for each pack instead of tracking all possible combinations of inequalities. This kind of partitioning is reminiscent of value partitioning, but with the important difference that variable packing and octagons operate on sets of program variables whereas value partitioning works on individual abstract values. In our work with analysis of JavaScript libraries, we have not encountered a critical need for tracking numeric relations.

The well-known analyzer Astrée [4] applies not only trace partitioning and octagons, but also a decision tree abstract domain that is used for tracking relations between booleans and numerical variables that affect control flow. That technique has some similarities with our type partitioning mechanism but relies on variable packing to avoid combinatorial explosions, whereas type partitioning uses the more lightweight value partitioning technique in combination with the existing control sensitivity mechanism of TAJS.

The main purpose of value partitioning is to be able to reason about relations between different parts of the abstract state (i.e., program variables and registers) at the various program points. Some literature uses the term relational analysis with a slightly different meaning: to relate information across program points, typically relations between the entry and exits of functions [8, 5].

Static analysis for JavaScript

Through the last decade, several static analyzers for JavaScript have been developed, including WALA [25, 24, 28], SAFE [17, 22], JSAI [13], and TAJS [11, 2, 26]. Although we focus on TAJS, the designs of SAFE and JSAI are reasonably similar, so we believe value partitioning could also be incorporated into those tools with little effort.

As discussed in the introduction, much work has been put into improving precision of the analyses through different kinds of context sensitivity and elaborate abstract domains. The techniques include parameter sensitivity and heap context sensitivity [2], loop unrolling [22],

[20] In our experiments, 99.4% of all abstract values have the trivial {ANY} partitioning.

[21] This was shown experimentally in the work on TAJS_{VR} [26, Section 7.1].

and syntactic patterns for detecting correlated read/write pairs and guiding context sensitivity [25]. Other works have explored more expressive string abstractions to reason more precisely about property names in dynamic property accesses [18, 1, 21]. Our abstract domain extension for value partitioning has few assumptions about the underlying abstract domain, so most of these techniques can be combined with value partitioning.

Despite such precision improvement techniques, imprecision is inevitable, and only a few techniques have been designed to handle dynamic property accesses with imprecise property names, most importantly, CompAbs-style trace partitioning [16] and demand-driven value refinement [26]. Previous work has shown that demand-driven value refinement enables analysis of many more challenging benchmarks than CompAbs-style trace partitioning (as also discussed in Section 6), and that the trace partitioning approach causes a large amount of redundant computation [26, Section 7.1]. The fundamental drawback of demand-driven value refinement is that it requires a separate backwards abstract interpreter for not only the entire JavaScript language but also the standard library. The backwards abstract interpreter of TAJS_{VR} is not simply the dual of TAJS but works goal-directed and with its own abstract domain based on intuitionistic separation logic. In contrast, value partitioning directly leverages the existing forward analyzer and thereby supports both the JavaScript language and the standard library essentially for free, which makes this approach substantially easier to develop and maintain. Furthermore, value partitioning is more general (for example, it enables type partitioning), and the three instantiations we have presented lead to better precision (for the Lodash4 tests).

The HOO (heap with open objects) abstract domain [9] is a relational abstraction that is designed to reason more precisely about abstract objects whose properties cannot be known statically. That approach is highly expressive but not scalable to real-world JavaScript libraries as those considered in Section 6.

8 Conclusion

We have presented value partitioning, a static analysis technique for reasoning about relational properties. It is a lightweight alternative to traditional trace partitioning techniques that allows relational information to be incorporated into the abstract values instead of requiring separate abstract states for the partitions. We have proposed three instantiations of value partitioning in JavaScript analysis: property name partitioning, free variable partitioning, and type partitioning, which enable precise reasoning for dynamic read/write pairs, free variables, and predicate functions, respectively.

The experimental results show that extending the TAJS analyzer with the three variants of value partitioning enables precise and efficient analysis of complex JavaScript libraries including Lodash and Underscore, thereby outperforming a state-of-the-art technique that relies on trace partitioning and without requiring a complicated backwards analysis. For the libraries considered in this study, property name partitioning has the largest effect among the proposed variants.

An interesting direction for future research is to investigate whether some of the traditional context sensitivity strategies used in TAJS and other JavaScript analyzers can be reformulated as new value partitioning instantiations, to make analysis faster while retaining precision.

References

1 Roberto Amadini, Alexander Jordan, Graeme Gange, François Gauthier, Peter Schachte, Harald Søndergaard, Peter J. Stuckey, and Chenyi Zhang. 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, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I*, volume 10205 of *Lecture Notes in Computer Science*, pages 41–57, 2017.

2 Esben Andreasen and Anders Møller. Determinacy in static analysis for jQuery. In *Proc. ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 17–31, 2014.

3 Esben Sparre Andreasen, Anders Møller, and Benjamin Barslev Nielsen. 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, Barcelona, Spain, June 18, 2017*, pages 31–36, 2017.

4 Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003*, pages 196–207. ACM, 2003.

5 Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. *J. ACM*, 58(6):26:1–26:66, 2011.

6 David R. Chase, Mark N. Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation (PLDI), White Plains, New York, USA, June 20-22, 1990*, pages 296–310. ACM, 1990.

7 Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

8 Patrick Cousot and Radhia Cousot. Modular static program analysis. In *Compiler Construction, 11th International Conference, CC 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*, volume 2304 of *Lecture Notes in Computer Science*, pages 159–178. Springer, 2002.

9 Arlen Cox, Bor-Yuh Evan Chang, and Xavier Rival. Automatic analysis of open objects in dynamic language programs. In *Static Analysis - 21st International Symposium, SAS 2014, Munich, Germany, September 11-13, 2014. Proceedings*, pages 134–150, 2014.

10 Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. Typing local control and state using flow analysis. In *Programming Languages and Systems - 20th European Symposium on Programming, ESOP 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*, volume 6602 of *Lecture Notes in Computer Science*, pages 256–275. Springer, 2011.

11 Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for JavaScript. In *Proc. 16th International Static Analysis Symposium*, pages 238–255, 2009.

12 John B. Kam and Jeffrey D. Ullman. Monotone data flow analysis frameworks. *Acta Inf.*, 7(3):305–317, September 1977.

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

14 Vineeth Kashyap, John Sarracino, John Wagner, Ben Wiedermann, and Ben Hardekopf. Type refinement for static analysis of JavaScript. In *DLS'13, Proceedings of the 9th Symposium on Dynamic Languages, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 17–26. ACM, 2013.

15 Gary A. Kildall. A unified approach to global program optimization. In *Conference Record of the ACM Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, October 1973*, pages 194–206. ACM Press, 1973.

16 Yoonseok Ko, Xavier Rival, and Sukyoung Ryu. Weakly sensitive analysis for JavaScript object-manipulating programs. *Softw., Pract. Exper.*, 49(5):840–884, 2019.

17 Hongki Lee, Sooncheol Won, Joonho Jin, Junhee Cho, and Sukyoung Ryu. SAFE: formal specification and implementation of a scalable analysis framework for ECMAScript. In *Proc. International Workshop on Foundations of Object Oriented Languages*, 2012.

18 Magnus Madsen and Esben Andreasen. String analysis for dynamic field access. In *Proc. 23rd International Conference on Compiler Construction*, volume 8409 of *Lecture Notes in Computer Science*. Springer, 2014.

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

20 Erik M. Nystrom, Hong-Seok Kim, and Wen-mei W. Hwu. Importance of heap specialization in pointer analysis. In *Proceedings of the 2004 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE'04, Washington, DC, USA, June 7-8, 2004*, pages 43–48. ACM, 2004.

21 Changhee Park, Hyeonseung Im, and Sukyoung Ryu. Precise and scalable static analysis of jquery using a regular expression domain. In *Proceedings of the 12th Symposium on Dynamic Languages*, DLS 2016, pages 25–36, New York, NY, USA, 2016. ACM.

22 Changhee Park and Sukyoung Ryu. Scalable and precise static analysis of JavaScript applications via loop-sensitivity. In *Proc. 29th European Conference on Object-Oriented Programming*, pages 735–756, 2015.

23 Xavier Rival and Laurent Mauborgne. The trace partitioning abstract domain. *ACM Trans. Program. Lang. Syst.*, 29(5), August 2007.

24 Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. Dynamic determinacy analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 165–174. ACM, 2013.

25 Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip. Correlation tracking for points-to analysis of JavaScript. In *Proc. 26th European Conference on Object-Oriented Programming*, 2012.

26 Benno Stein, Benjamin Barslev Nielsen, Bor-Yuh Evan Chang, and Anders Møller. Static analysis with demand-driven value refinement. *Proceedings of the ACM on Programming Languages (PACMPL)*, 3:140:1–140:29, 2019.

27 Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, 13(2):181–210, 1991.

28 Shiyi Wei, Omer Tripp, Barbara G. Ryder, and Julian Dolby. 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, Seattle, WA, USA, November 13-18, 2016*, pages 487–498. ACM, 2016.