Desynchronized Multi-State Abstractions for Open Programs in Dynamic Languages

Arlen Cox[1,2], Bor-Yuh Evan Chang[1], and Xavier Rival[2]

[1] University of Colorado Boulder, {arlen.cox,evan.chang}@colorado.edu
[2] INRIA/CNRS/ENS Paris, xavier.rival@ens.fr

Abstract. Dynamic language library developers face a challenging problem: ensuring that their libraries will behave correctly for a wide variety of client programs without having access to those client programs. This problem stems from the common use of two defining features for dynamic languages: callbacks into client code and complex manipulation of attribute names within objects. To remedy this problem, we introduce two state-spanning abstractions. To analyze callbacks, the first abstraction desynchronizes a heap, allowing partitions of the heap that may be affected by a callback to an unknown function to be frozen in the state prior to the call. To analyze object attribute manipulation, building upon an abstraction for dynamic language heaps, the second abstraction tracks attribute name/value pairs across the execution of a library. We implement these abstractions and use them to verify modular specifications of class-, trait-, and mixin-implementing libraries.

1 Introduction

"Don't Repeat Yourself!" This DRY mantra leads JavaScript developers to minimize the code that they write and thus minimize the number of places bugs can occur. As a result, there is a proliferation of generic libraries and code reuse in the JavaScript community. Unfortunately, even though library authors would like to know that their libraries work correctly with any client, current verification techniques cannot verify this because they do not also follow the DRY mantra – they require reverifying libraries along with each and every client [15, 17–19, 27]. This paper brings the DRY mantra to automatic dynamic language verification by modularly verifying libraries *without the presence of client code*.

While there are many kinds of libraries for many dynamic languages, this paper focuses on meta-feature libraries for JavaScript. *Meta-feature libraries* add functionality that is commonly built-in to languages, such as mixins, traits, classes, and memoization. These features are not first-class features of the JavaScript language, but they aid software engineering, so nearly every program includes them in some form or another. For example, the ubiquitous jQuery, Prototype, and MooTools libraries all include implementations of mixins. Similarly, MooTools, Prototype, and the Microsoft Ajax Library include class implementations. What makes these libraries unique is their use of open object manipulation, functions, and encapsulation to implement language features as libraries.

For example, while JavaScript does not contain classes, a simple version of classes can be implemented with the few lines shown in Figure 1. These few lines implement classes by constructing a class instantiation function (highlighted) that is responsible for creating new instances of the class. This class instantiation function is derived from a configuration object

```
1  var Class = function(cfg) { //make class
2    var copy = function(src,exc) {...};
3    var attrs = copy(cfg,{});
4    var init = cfg.init;
5    return function(args) { //make instance
6      var result = copy(attrs,{init:null});
7      init(result, args);
8      return result;
9    };
10 }
```

Fig. 1: `Class` implements a simple version of classes. The class is essentially allocated by the call to `copy` on line 3. The instance is allocated by the call to `copy` on line 6. Line 7 calls the initialization function on the instance.

`cfg` that describes not only a template for the instance object, but an initialization function `init`. The `init` function is run on each newly created instance, completing the initialization of the new object using arguments passed to the instantiation function. Note that because JavaScript allows the attributes of objects to be mutated (i.e. objects are *open*), it is necessary to copy the configuration object twice to create an instance. The first copy (underlined) creates a backup that ensures that if the configuration object is mutated, already constructed classes are not mutated as well. The second copy (line 6) creates the instance object.

A key challenge of verifying library implementations is that developers specify libraries in terms of input/output behaviors. If a particular kind of input is given, a particular, but related kind of output is given. For example, in `Class`, the object generated by instantiating a class is related to the `cfg` object that was passed in to `Class`. This means inputs to a library must be treated as unknowns that can be related to the outputs of that library — even when the inputs are *unknown functions* or objects with *unknown attribute name/value relationships*.

The core problem with unknown functions (such as `init`) as input to a library is that they may be called by the library. If they are, they may have wide-reaching effects on the state of the program. However, developers are not stymied by these function calls when reading code because the effects are usually well contained by the surrounding code. Developers use conventions such as copying into local, non-escaping variables (like `attrs` and `init`) to ensure that certain parts of the program's state cannot be affected by calls to unknown functions. Therefore, when developers are reasoning about this code, they optimistically assume that when a call to one of these unknown functions occurs, there are two parts of the program memory: (1) the part unaffected by the call, which may be freely accessed and modified after the call and (2) the part affected by the call, which, over the remainder of the function is solely described as "the result of calling the function on whatever that part was before the call." In this paper, we observe that analyses that are designed for such library code can optimistically split the heap into two parts, where the analysis can proceed on the unaffected part and the affected part can be saved along with the function that affected it until that function is known.

Furthermore, existing analyses have problems with input objects that have unknown attribute name/value relationships. Most analyses represent containers by partitioning them.

However only using partitions, it is not possible to represent the fact that attribute/value pairs are often preserved. For example, when `cfg` is copied to `attrs`, it is clear that every attribute/value pair is copied and therefore, all attribute/value pairs are preserved as-is across this computation. As Halbwachs and Péron [14] discovered for arrays, it is beneficial to capture relations between individual attributes and values and to share those relations between multiple containers. However, these relations can be generalized beyond arrays to any container and can be extended to relate partitions across multiple states. This allows proving that `attrs` is equal to what `cfg` was at the beginning of the class creation.

To verify modular specifications of JavaScript libraries, even when client code is absent, and thus enabling reuse of specifications, and improving library reliability, we make the following contributions:

- To abstract open objects and containers with unknown attribute name/value relationships, we introduce *attribute/value trackers* that extend existing container and open-object abstractions with the ability to perform fully precise partitioning when attributes and values are copied. Trackers represent a form of parametric polymorphism for attribute/value relationships that can be applied across multiple abstract heaps to relate unknown input objects to unknown output objects.
- For the analysis of a call to an unknown function, we introduce *desynchronized separation*, which splits off a region of the heap by representing it as an old analysis state along with the code required to synchronize that portion of the state with the rest of the analysis. This creates a form of assume-guarantee reasoning that mimics the programmer intuition for simple, well-contained callbacks, while enabling automatic analysis.
- We extend the heap with open objects abstraction (HOO) with attribute/value trackers and desynchronized separation and evaluate these additions to HOO by automatically verifying specifications written for JavaScript meta-feature libraries. We analyze the core functionality of libraries that implement mixins, traits, classes, and memoization. By utilizing HOO along with both desynchronization and attribute/value trackers, we are able to fully precisely analyze these library cores, even without any knowledge of specific attribute names used in input objects or code for client-supplied callbacks.

2 Overview

In this section we demonstrate the power of attribute/value trackers and desynchronized separation applied to HOO (the Heap with Open Objects Abstraction [8]) by showing key parts of the analysis of instantiating a class created by the `Class` library introduced in Figure 1. First we show how attribute/value trackers enhance open-object and container abstractions with the analysis of the `copy` function used by `Class`. Then, we show how desynchronization allows analysis of calls to unknown functions.

2.1 Preliminaries

Before we explain attribute/value trackers and desynchronized separation, we introduce the basics of the HOO abstraction. HOO is a separation-logic-based abstraction for dynamic language heaps that supports reasoning about open objects, which behave like containers

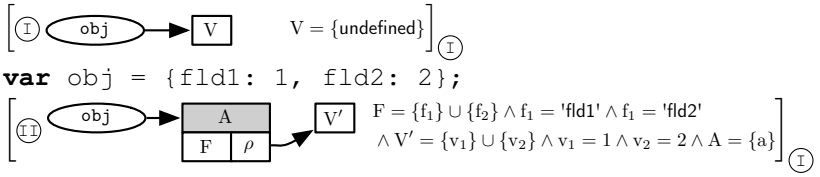


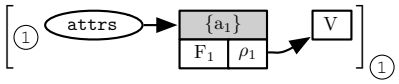
Fig. 2: The HOO abstract domain represents a heap of open objects using a combination of a heap graph and pure side constraints

mapping strings representing attribute names to values. HOO supports the basic requirements for both attribute/value trackers and desynchronization. It partitions open objects by the attributes (as most container abstractions do) and it supports partitioning the heap (as all separation-logic-based abstractions do). What makes HOO unique is its use of a set abstraction to relate partitions to one another. However, this functionality is not strictly required to make use of trackers and desynchronized separation.

Because we are concerned with input/output relationships, Figure 2 shows a simple program annotated with two-state HOO invariants. The first invariant Ⓘ shows the initial heap containing the variable `obj` pointing to the value undefined. The input heap is indicated with its program point in the lower right hand corner of an invariant. In the case of Ⓘ, the input heap is the same as the heap shown in brackets at Ⓘ. The current heap, relative to that input heap is shown in the brackets along with a constraint on the logic variables used in both heaps. This constraint is represented and manipulated by an abstract domain for sets.

This program creates a new object pointed to by `obj` that has two attributes: 'fld1' corresponds to the value 1 and 'fld2' corresponds to the value 2. The abstract state Ⓘ highlights the important parts of the abstraction. The heap part in brackets shows an abstract object that is represented as a table. The shaded top row is the set symbol A for the base address of objects. If this is not a singleton set, the object is a summary. On the right, A is constrained to be a singleton set of addresses and thus it is not a summary object. Below the shaded top row are rows each describing a partition of attribute names for that object. Here we have decided to represent these two attribute names 'fld1' and 'fld2' using a single partition that conflates the two attribute names. This partition is represented with the set symbol F, where it is equated to the union of two singleton sets with attribute names f_i. Additionally, this partition has been assigned the attribute/value tracker ρ, which can keep track of specific attribute/value pairs from the beginning of the function to the end, as will be demonstrated in next section. Finally, the partition points to a set of values V' that is made up of individual values v_i. Note that this is not the most precise abstraction because the two attributes have been summarized into a single partition. An alternative abstraction would construct a separate partition for each known attribute name.

In this paper we will often use a shorthand notation where instead of showing a set symbol such as A in the heap, we will show instead a singleton set in brackets, such as $\{\mathrm{a}\}$. This is equivalent to having a set symbol and then constraining that set symbol to be equal to the singleton set. This is useful for improving the readability of the notation, but formally all symbols in the heap are set symbols.




```
var exc = {init:1};
var result = {};
for(var a in attrs) {
  if(attrs.hasOwnProperty(a) && !exc.hasOwnProperty(a)) {
    result[a] = attrs[a];
  }
}
```

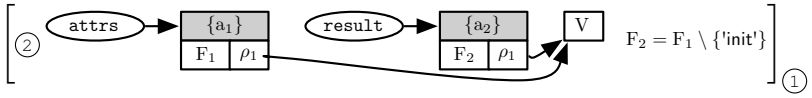


Fig. 3: Analysis of class instantiation uses attribute/value trackers to maintain precision when attributes are copied

2.2 Attribute/Value Trackers

At the start of the analysis of the class instantiation function (the highlighted part of Figure 1), the first code that the analysis encounters is the call to the `copy` function. Figure 3 shows the body of the `copy` function after it has been inlined into the context of the class instantiation. This function iteratively copies one open object `attrs` to another open object `result` by first checking if the attribute name that is being copied is in the exclusion object `exc`. Accompanying the `copy` function are pre/postconditions that show a portion of heap that is relevant to this function.

An abstraction such as HOO does a nice job of incrementally inferring the relationship that forms between the `result` object and the `attrs` object. While, as a two-state abstraction, HOO can relate initial objects to final objects, it still conflates all of the attributes and values that may have been in that object into a single partition. This means that while HOO can prove that the `result` object has a subset of the attributes of the initial `attrs` object, it cannot prove by itself that the attribute/value relationship was maintained for everything that was copied. This is where attribute/value trackers come in.

An attribute/value tracker is an *uninterpreted symbol* for some relationship between attributes and values. When a tracker is applied to a particular partition and corresponding values, it uses that "global" relationship to constrain exactly which values can possibly correspond to which attributes that are described by the partition. The most important aspect of an attribute/value tracker is that it is "global" in the sense that the symbol is shared between the two-states of the invariant. A tracker's meaning is consistent across these two abstract heaps.

Throughout this analysis, there is only one attribute/value tracker ρ_1. In the precondition the attribute/value tracker ρ_1 can be automatically added, as at that point the true relationship between attributes and values is unknown. But in the postcondition, the fact that ρ_1 is used for two partitions means not only that `attrs` and `result` have the same attribute and value relationship after the loop, but that the relationship is the same one that existed before the loop.

Critically, once a tracker is associated with a partition, that tracker can be reused with any other partition that is a subset of that initial, associated partition. Here, we see that the same tracker ρ_1 is used in the F_2 partition of the object at address a_2. Even though the F_2 partition

Fig. 4: Materialization maintains the attribute/value tracker ρ_1

is a subset of F_1 used in the object at address a_1, the same tracker can be used. As a result, this constraint says that the `result` object is *exactly* the same as the `attrs` object except that the 'init' attribute has been removed if it was present.

Materialization with Attribute/Value Trackers: In the loop body, before the object pointed to by `attrs` can be read, the single attribute that will be read must be materialized in that object. This ensures strong updates occur. An example materialization is shown in Figure 4. On the left is the object at address a_1 before materialization and on the right is the same object after materialization. Here, we assume that the particular attribute is represented by the symbol f, and while f is not explicitly constrained, it is known that f is one of the attributes from F_1.

What is special about attribute/value trackers is that rather than requiring a new description of the partition when a materialization occurs; here they can be duplicated. On the right the tracker ρ_1 occurs in both partitions F_1' and $\{\mathrm{f}\}$. This is because the tracker only restricts the values that correspond to those in the partition. Since the partition has been refined, the same restriction can be applied to both new partitions.

Transfer of Attribute/Value Trackers: As part of analyzing code like `copy` above, there is a transfer of an attribute/value pair from one object to another. This transfer maintains the relationship between attributes and values. When transfer occurs, the attribute/value tracker can be transfered along with the attribute and value. Therefore, even if the particular attribute and particular value cannot be identified from their sets, the tracker maintains whatever the original relationship was and allows it to be transfered to other objects.

Here this property of trackers ensures that ρ_1 is transfered from the `attrs` object to the `results` object. Since the transfer occurs whenever the attribute/value pair is copied, the tracker can be unconditionally copied. However, because the resulting partition F_2 is restricted, this simply limits the scope of where the tracker can be applied.

While we have not demonstrated the use of summaries generated by this analysis, attribute/value trackers are critical to this application. With the use of attribute/value trackers, a general precondition can be specialized for a particular calling context, essentially a form of parametric polymorphism. That is, partitions can be more finely specified corresponding to the actual objects passed into library functions. This very same partitioning can be applied to postcondition, allowing precision that was made available well after the analysis was completed to be preserved by the analysis.

2.3 Desynchronized Separation

When analysis reaches the call to the client-supplied initializer that is shown in Figure 5, there is a problem. The actual function that is called is an input to the class library and as a result it is unknown to the analysis. However, despite the fact that this function is unknown, developers might optimistically reason about what this class library does as follows: `attrs` is protected by lexical scoping, so it should not change, and `result` is initialized by the copy and then

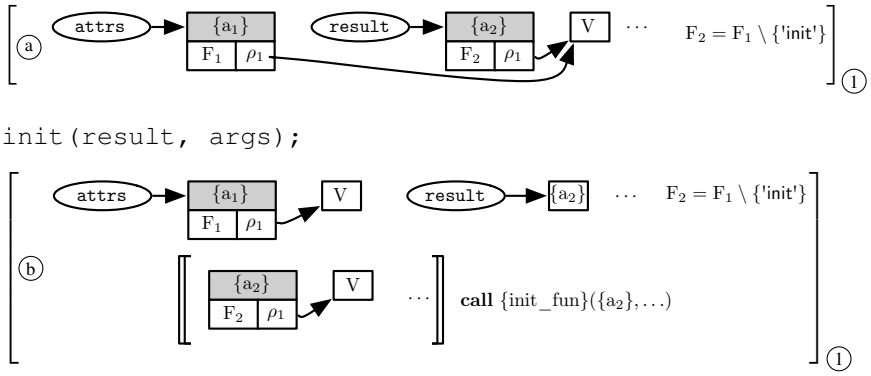


Fig. 5: Desynchronized terms are introduced by function calls to unresolvable functions

the return value is whatever `result` is after running the client-supplied initializer `init` on it. Desynchronized separation is a means for capturing this kind of optimistic reasoning in a sound manner using a form of assume-guarantee reasoning.

Immediately before the call to the initializer, there are two objects shown: a_1 is the `attrs` object, which is the backup copy of the `cfg` object that was passed in to `Class` and a_2 is the `result` object that is the class instance that is currently in the process of being constructed. The relationship between F_1 and F_2 carries over from the copy as before. Other parts of the heap are not shown, as they are not necessary for explaining desynchronized separation.

When the analysis reaches the call to `init`, desynchronized separation optimistically splits the heap into two separate parts: (1) the part that shall not be used by the client-supplied initializer and (2) the part that shall be used by the client-supplied initializer. In our algorithm, we make this split based on reachability: optimistically *assuming* the post-call code in the caller does not use anything reachable from the arguments to the call. Thus for the unused portion, there is no change and thus it is directly represented in the post-state ⓑ. For the used portion, the function call may have changed it and thus it is desynchronized. Desynchronization represents the resulting heap as a term that stores the used portion of the heap before the call and the function that is applied.

The desynchronization process introduces a *desynchronized term*, written $[\![H]\!]\mathbf{call}\ \mathrm{V}(...)$, where H is the portion of the heap that is desynchronized and $\mathbf{call}\ \mathrm{V}(...)$ is the function called and the arguments passed to it. By introducing this desynchronized term, the post-state of the call can be written in such a way that, when the client-supplied initializer becomes known, such as when a function summary generated by HOO is reused, the now known function can convert the desynchronized portion back into a normal, "synchronized" heap formula.

In ⓑ we can see that the heap has been split so that a_2 has been desynchronized. Because it may have been modified by the call, it is "locked" in a state from before the function call. That is, we *guarantee* in the analysis that the post-call code does not access desynchronized sub-heaps by ensuring the analysis gets stuck (raises a warning) if accessing desynchronized memory. Desynchronization is different from simply separating the two parts of the heap because the desynchronized region represents the portion of the heap that results from calling the desynchronized function on the desynchronized part of the heap. In this way it soundly

abstracts calls to unknown functions by explicitly representing the precondition to the call and implicitly representing the postcondition of the call.

A significant part of implementing desynchronized separation is the operation used to split the heap into the desynchronized and non-desynchronized parts. In this paper we outline a simple means of splitting the heap based on reachability as in [25] that exploits the fact that JavaScript developers, by convention, protect regions of the heap using closures for encapsulation. Here, `attrs` is protected in such a way. Consequently, the heap split that is automatically inferred leaves a_1 and all local variables outside the desynchronized term and places a_2 inside the desynchronized term. With this split, it is possible to verify that `attrs` is unmodified by class instantiation, which means that classes are immutable and it is possible to verify that the object built in the class is the one returned by the class after calling the client-supplied initializer on it starting from elements copied from `attrs`.

3 Abstracting Callbacks and Objects with Multi-State Abstraction

In this section we define attribute/value trackers and desynchronization as an extension to the heap with open objects (HOO) abstract domain [8]. First we present attribute/value trackers and how they are added to HOO. Then we present desynchronized separation, also adding it to HOO.

Throughout these sections we utilize the following symbols in the definitions.

$$\begin{array}{ll}
\overline{\mathrm{Address}} \subseteq \overline{\mathrm{Value}} & d \subseteq \overline{\mathrm{Attribute}} \\
\overline{\mathrm{Attribute}} \subseteq \overline{\mathrm{Value}} & o \in \overline{\mathrm{Object}} = \overline{\mathrm{Attribute}} \rightharpoonup \overline{\mathrm{Value}} \\
v \in \overline{\mathrm{Value}} & \sigma \in \overline{\mathrm{State}} = \overline{\mathrm{Address}} \rightharpoonup \overline{\mathrm{Object}} \\
f \in \overline{\mathrm{Attribute}} &
\end{array}$$

$\overline{\mathrm{Address}}$ is the set of all concrete addresses, $\overline{\mathrm{Attribute}}$ is the set of all concrete attributes (strings), and $\overline{\mathrm{Value}}$ is the set of all values including addresses and attributes. $\overline{\mathrm{Object}}$ is the set of partial functions from attributes to values, where unmapped attributes are not attributes in the object. Similarly concrete states are a partial function from addresses to objects. Individual concrete values v, attributes f, and object domains d are used in defining semantics.

3.1 Attribute/Value Trackers on HOO

Attribute/value trackers extend an existing domain for containers that supports strong updates. Attribute/value trackers significantly increase the precision of the existing container domains by precisely keeping track of the relationship between individual attributes and individual values, even when the container has summarized many attributes and values into a single partition. An *attribute/value tracker* is an uninterpreted partial function ρ that is optionally added to each container partition in an existing abstract domain for containers.

HOO is a separation-logic-based approximation of a heap that is restricted by an abstraction for sets of values. This abstraction for sets restricts relationships between symbols each representing a set:

$$\{\mathrm{a}\},\{\mathrm{f}\},\{\mathrm{v}\},\mathrm{A},\mathrm{F},\mathrm{V} \in \overline{\mathrm{Symbol}}$$

where A represents a set of addresses, F represents a set of attributes, and V represents a set of values. The $\{\mathrm{a}\}$, $\{\mathrm{f}\}$, and $\{\mathrm{v}\}$ sets are the respective singleton forms.

Definition 1 (Attribute/Value Trackers with HOO). *The heap with open objects abstract domain, when extended with attribute/value trackers, is represented with the following logical syntax:*

$$\begin{aligned}\widehat{\mathrm{Heap}} \ni H &::= H_1 * H_2 \mid \mathrm{A}\cdot\langle O\rangle \mid \textsc{Emp} \mid \textsc{True}\\ \widehat{\mathrm{Object}} \ni O &::= O_1 ; O_2 \mid \mathrm{F} : \rho \mapsto \mathrm{V} \mid \mathrm{F} : - \mapsto \mathrm{V} \mid \textsc{None}\\ \widehat{\mathrm{Domain}} \ni D &::= D_1 \vee D_2 \mid [H_2]_{H_1} \mathbin{¦} P\end{aligned}$$

An abstract state D is either a disjunction of abstract states, or a triple $[H_2]_{H_1} \mathbin{¦} P$ representing an initial heap H_1 and a current heap H_2 restricted by a domain instance P for sets. The domain responsible for representing P is a parameter to this abstraction and unspecified. An individual heap H is a standard separation logic heap consisting of two disjoint parts combined with separating conjunction, a set of objects $\mathrm{A}\cdot\langle O\rangle$ at addresses described by A *with structure O, or the empty* Emp *or unknown* True *heap. Objects are a form of container, which is represented by a number of disjoint partitions of the attributes. A single partition is represented as either $\mathrm{F} : \rho \mapsto \mathrm{V}$ or $\mathrm{F} : - \mapsto \mathrm{V}$ depending on whether the attribute/value tracker ρ is present or not. Partitions are joined together into objects using another form of separating conjunction* ; *whose unit is the empty object* None*.*

Figure 6 shows that an instance of HOO concretizes to a set of pairs of concrete states along with a valuation. The σ_0 state represents a starting state for a library function and the σ_1 state represents the current state relative to σ_0. The valuation maps each symbol that occurs in the heap formula, including those representing sets of addresses, attributes and values to a set of concrete addresses, attributes, or values:

$$\eta : \overline{\mathrm{Valuation}} = \overline{\mathrm{Symbol}} \to \mathcal{P}\left(\overline{\mathrm{Value}}\right)$$

The valuation ensures that symbols map to consistent values throughout a concretization, even if the symbol is used multiple times. The concretization of P produces a set of these valuations as must be defined by the abstraction for sets. The concretization for any instance of the abstraction for sets must have the following type.

$$P \in \widehat{\mathrm{Sets}} \qquad\qquad \gamma_P : \widehat{\mathrm{Sets}} \to \mathcal{P}\left(\overline{\mathrm{Valuation}}\right)$$

For the concretization of heaps and objects, there is an additional value that is returned besides the valuation η and the state σ. The attribute/value tracker map μ binds trackers to their corresponding partial functions:

$$\begin{aligned}&\rho \in \overline{\mathrm{TrackSym}}\\ &\mu \in \overline{\mathrm{TrackerMap}} = \overline{\mathrm{TrackSym}} \rightharpoonup \overline{\mathrm{Attribute}} \rightharpoonup \overline{\mathrm{Value}}\end{aligned}$$

An element $\mu \in \overline{\mathrm{TrackerMap}}$ maps a tracker symbol to a partial function from attributes to values. The domain of that function is fixed when the tracker is introduced (Section 4.2).

$$\gamma : \widehat{\mathrm{Object}} \to \mathcal{P}\left(\overline{\mathrm{Valuation}} \times \overline{\mathrm{TrackerMap}} \times \overline{\mathrm{Object}} \times \mathcal{P}\left(\overline{\mathrm{Attribute}}\right)\right)$$

$$\gamma(O_1;O_2) = \left\{ \eta,\mu,o,d \;\middle|\; \begin{array}{l} \exists o_1,o_2,d_1,d_2.\, (\eta,\mu,o_1,d_1) \in \gamma(O_1) \wedge (\eta,\mu,o_2,d_2) \in \gamma(O_2) \\ \quad \wedge o = o_1 \cup o_2 \wedge d = d_1 \uplus d_2 \wedge \mathrm{Dom}(o_1) \cap \mathrm{Dom}(o_2) = \emptyset \end{array} \right\}$$

$$\gamma(\mathrm{F} : \rho \mapsto \mathrm{V}) = \left\{ \eta,\mu,o,d \;\middle|\; \begin{array}{l} d = \eta(\mathrm{F}) \wedge \forall f \in \eta(\mathrm{F}). \\ \quad o(f) \in \eta(\mathrm{V}) \wedge \mu(\rho)(f) = o(f) \end{array} \right\}$$

$$\gamma(\mathrm{F} : - \mapsto \mathrm{V}) = \left\{ \eta,\mu,o,d \;\middle|\; d = \eta(\mathrm{F}) \wedge \forall f \in \eta(\mathrm{F}).\, o(f) \in \eta(\mathrm{V}) \right\}$$

$$\gamma(\textsc{None}) = \{\eta,\mu,[],\emptyset\}$$

$$\gamma : \widehat{\mathrm{Heap}} \to \mathcal{P}\left(\overline{\mathrm{Valuation}} \times \overline{\mathrm{TrackerMap}} \times \overline{\mathrm{State}}\right)$$

$$\gamma(H_1 * H_2) = \left\{ \eta,\mu,\sigma \;\middle|\; \begin{array}{l} \exists \sigma_1,\sigma_2.\, (\eta,\mu,\sigma_1) \in \gamma(H_1) \wedge (\eta,\mu,\sigma_2) \in \gamma(H_2) \\ \quad \wedge \sigma = \sigma_1 \cup \sigma_2 \wedge \mathrm{Dom}(\sigma_1) \cap \mathrm{Dom}(\sigma_2) = \emptyset \end{array} \right\}$$

$$\gamma(\mathrm{A} \cdot \langle O \rangle) = \left\{ \eta,\mu,\sigma \;\middle|\; \begin{array}{l} \forall a \in \eta(\mathrm{A}).\, \exists o,d. \\ \quad \sigma(a) = o \wedge (\eta,\mu,o,d) \in \gamma(O) \wedge \mathrm{Dom}(o) = d \end{array} \right\}$$

$$\gamma(\textsc{Emp}) = \{\eta,\mu,[]\}$$

$$\gamma(\textsc{True}) = \overline{\mathrm{Valuation}} \times \overline{\mathrm{TrackerMap}} \times \overline{\mathrm{State}}$$

$$\gamma : \widehat{\mathrm{Domain}} \to \mathcal{P}\left(\overline{\mathrm{Valuation}} \times \overline{\mathrm{State}} \times \overline{\mathrm{State}}\right)$$

$$\gamma(D_1 \vee D_2) = \left\{ \eta,\sigma_1,\sigma_2 \;\middle|\; (\eta,\sigma_1,\sigma_2) \in \gamma(D_1) \vee (\eta,\sigma_1,\sigma_2) \in \gamma(D_2) \right\}$$

$$\gamma([H_2]_{H_1} ! P) = \left\{ \eta,\sigma_1,\sigma_2 \;\middle|\; \begin{array}{l} \exists \mu.\, (\eta,\mu,\sigma_1) \in \gamma(H_1) \\ \quad \wedge (\eta,\mu,\sigma_2) \in \gamma(H_2) \wedge \eta \in \gamma(P) \end{array} \right\}$$

Fig. 6: Concretization of HOO abstract states along with attribute/value trackers

Example 1 (Attribute/Value Trackers with HOO). In the following state, there are two abstract heaps and a single pure domain instance.

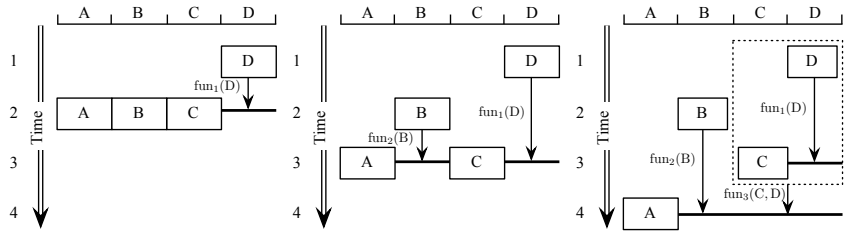
$$[\{\mathrm{a}\} \cdot \langle \mathrm{F}' : \rho \mapsto \{\mathrm{v}\} \rangle]_{\{\mathrm{a}\} \cdot \langle \mathrm{F} : \rho \mapsto \{\mathrm{v}\} \rangle} ! \mathrm{F}' \subseteq \mathrm{F}$$

This constrains the relationship between the pre-state and the current state so that they both refer to the same object because they use the same symbol $\{\mathrm{a}\}$ and the number of attributes has been possibly reduced: some attributes may have been deleted. All other attributes remain the same and no attributes can have been observably added (added and then later removed is acceptable).

Additionally, the attribute/value tracker ensures that the partition F' is exactly the same as F except for the elements that are removed.

3.2 Desynchronized Separation

Desynchronized separation is an extension to a separation logic that adds a desynchronized term to the logical formulas. It is useful for representing different parts of the heap from



(a) After call to $\mathrm{fun}_1(\mathrm{D})$ (b) After call to $\mathrm{fun}_2(\mathrm{B})$ (c) After call to $\mathrm{fun}_3(\mathrm{C,D})$

Fig. 7: Three separate desynchronizations after calling three successive functions on four regions of memory. In (c), A is the current analysis state where as regions B, C, and D have all been desynchronized. The D region has been desynchronized twice.

different times during an analysis. As a result, it allows a meaningful representation of the heap after a call to an unknown function has been made.

Example 2 (Desynchronization). To demonstrate the power of desynchronization, Figure 7 shows the process of desynchronization pictorially. The program being considered has four separate regions of memory A, B, C, and D that are entirely self contained (no pointers between regions) and the program is about to evaluate three function calls whose bodies are unknown in sequence: $\mathrm{fun}_1(\mathrm{D})$; $\mathrm{fun}_2(\mathrm{B})$; $\mathrm{fun}_3(\mathrm{C,D})$. Figures 7 (a), (b), and (c) show the state of desynchronization after each of these calls. Initially, at time 1, all memory is synchronized and represented at time 1.

When analyzing the call to $\mathrm{fun}_1(\mathrm{D})$, the body is unknown and thus the analysis cannot continue. However, because the function can only affect the memory region D, it is possible to proceed if we desynchronize the heap. The result of the desynchronization is shown in Figure 7a. Regions A, B, and C are allowed to proceed on to time 2, but region D stays locked at time 1 and becomes inaccessible. This inaccessibility is critical because any of that memory in region D may have been mutated by the call to fun_1, and without any knowledge of what fun_1 did, it is impossible to say what the effect of accessing such memory would be.

Even though D has been desynchronized, we can still know a lot about the region after the function has been evaluated. Specifically, we can save which function was supposed to be evaluated, thus we know not only the state of the program before the function call, but we know the function call. With this information, if the function body were provided later, we could resynchronize D with A, B, and C by applying the analysis to that function body starting from D.

In Figure 7b we show the result after the call to $\mathrm{fun}_2(\mathrm{B})$. The only accessible region is B and thus it is desynchronized from the A and C regions. Because D is still inaccessible, it just becomes farther in time from being synchronized, but it is no more challenging to resynchronize it. Because B and D are completely distinct regions, there is no affect on B (or A or C) when resynchronizing D and thus even though B and D were desynchronized at different times, the resynchronization is no different.

Finally, in Figure 7c we show the result after the call to $\mathrm{fun}_3(\mathrm{C,D})$. Because it is possible that the result of region D is accessed here, the same region must be desynchronized again.

We show this nested desynchronization in the dashed box. Both C and D are desynchronized from A, which D is also now desynchronized from C.

To resynchronize everything after Figure 7c, the three functions must be evaluated. However, the order in which the functions are evaluated is irrelevant. Evaluating $\mathrm{fun}_1(\mathrm{D})$ first would resynchronize D with C (but not with A). Evaluating $\mathrm{fun}_2(\mathrm{B})$ first would resynchronize B with A. Evaluating $\mathrm{fun}_3(\mathrm{C},\mathrm{D})$ first would resynchronize C with A and would allow D to be resynchronized with A by only evaluating $\mathrm{fun}_1(\mathrm{D})$.

Definition 2 (Desynchronized Separation). *Desynchronized separation extends the logic presented in HOO with a desynchronizing term, an extra kind of heap H that represents a desynchronized portion of the heap along with the function to call and the arguments to pass to resynchronize that portion of the heap with the surrounding heap. The heap H now has the following grammar:*

$$\widehat{\mathrm{Heap}} \ni H ::= [\![H]\!]\mathbf{call}\ \mathrm{V}_f(\mathrm{V}_1,...,\mathrm{V}_n) \mid ...$$

To define the concretization of a desynchronized term, concrete values must be extended with functions. We do not give any specific semantics to these functions, but we do assume that while they can mutate the heap, they can only mutate the portion of the heap reachable from global variables, local variables or any closed variables. Essentially, the functions adhere to the standard framing conditions of separation logic [24]. The evaluation of a function is described by the relation

$$\langle\sigma\rangle\mathbf{call}\ v(v_1,...,v_n)\langle\sigma'\rangle$$

which evaluates a call to the function v starting from state σ, passing arguments v_1 to v_n and results in state σ'. Note that we assume all variables have been resolved to values before evaluating this function and thus no environment is necessary to express this computation. This minimizes the reachable heap, which may reduce the footprint of the desynchronized term.

The concretization of HOO with desynchronization is defined as an extension to the concretization of HOO. Because the signature of the function is not required to change, we only define the concretization of the new desynchronized terms:

$$\gamma([\![H]\!]\mathbf{call}\ \mathrm{V}_f(\mathrm{V}_1,...,\mathrm{V}_n)) \stackrel{\text{def}}{=} \left\{ \eta,\mu,\sigma \;\middle|\; \begin{array}{l} (\eta,\mu,\sigma_o)\in\gamma(H) \wedge v\in\eta(\mathrm{V}_f) \\ \wedge (v_1,...,v_n)\in\eta(\mathrm{V}_1)\times...\times\eta(\mathrm{V}_n) \\ \wedge \langle\sigma_o\rangle\mathbf{call}\ v(v_1,...,v_n)\langle\sigma\rangle \end{array} \right\}$$

The γ function concretizes the embedded heap H to a pre-state σ_o and its corresponding valuation. Then for each possible concrete value of the function and each argument, the state σ is the result of evaluating that function on those arguments starting from σ_o. Of course, what makes it possible to reason about applying a function to a portion of the heap is separating conjunction. This dictates that the portion of the heap σ_o was disjoint from the rest of the heap when the desynchronization was created and thus, after this call to a possibly unknown function, σ must be disjoint from the rest of the heap as well.

4 Analysis Using Multi-State Abstraction

In this section we formalize analysis using HOO with desynchronized separation and attribute/-value trackers. Because most of the JavaScript language has little effect on desynchronization

or attribute value trackers, we focus on the analysis of two core commands. Other commands are either critical to HOO (loops and branches) and documented in [8] or are not critical to any of these analyses. The two core commands are:

$$c ::= \mathbf{call}\ x(y_1,...,y_n) \mid x_1[x_2] := x_3[x_4]$$

The first command is a call to a function, where the function has been closure converted. We assume the corresponding closure and the global object are passed as arguments. The second command is responsible for copying an attribute/value pair from one object to another (handling missing attributes appropriately).

Analysis using HOO is standard abstract interpretation [7]. It infers invariants for each point in the program. Because HOO is a heap abstraction, each command in the language mutates the heap graph, but does not mutate the pure set abstraction P. Destructive updates are achieved by swinging pointers to fresh symbols and constraining those fresh symbols in P.

HOO's inclusion checking, join, and widening algorithms involve an object matching procedure where variables are matched, then objects pointed to by those variables are correspondingly matched. Within each of those objects, partitions are matched. This matching process proceeds summarizing objects from the same allocation site until all objects are matched (and summarized).

Inclusion checking: When performing an inclusion check such as the following, there are two kinds of mappping. The address mapping $M : \overline{\text{Symbol}} \to \overline{\text{Symbol}}$ maps each object symbol from the left-hand side to an object symbol from the right-hand side. Whereas the attribute mapping $J : \mathcal{P}\left(\mathcal{P}\left(\overline{\text{Symbol}}\right) \times \overline{\text{Symbol}}\right)$ is a set of sets of attribute partitions from the left-hand side and the corresponding attribute partition from the right-hand side.

$$H \backslash P \sqsubseteq^{J}_{M} H' \backslash P'$$

For each matched partition $(\bar{\text{F}},\text{F}) \in J$ if each $\text{F}_i \in \bar{\text{F}}$ is included in F, the inclusion check can hold. Otherwise it fails. Similarly, for each $\text{A}_1 \mapsto \text{A}_2 \in M$, if A_1 is included in A_2, the inclusion check can hold.

Join and widening: When performing join or widening the underlying operation is similar. The objects must be matched. The difference between the two algorithms is that the widening algorithm makes use of the underlying widening algorithm for pure operations and may produce different matchings in order to ensure analysis convergence. There are three kinds of matching for the following join: (1) $M_1 : \overline{\text{Symbol}} \to \overline{\text{Symbol}}$ is a mapping from the left-hand side to the result object; (2) $M_2 : \overline{\text{Symbol}} \to \overline{\text{Symbol}}$ is a mapping from the right-hand side to the result object; and (3) $J : \mathcal{P}\left(\mathcal{P}\left(\overline{\text{Symbol}}\right) \times \mathcal{P}\left(\overline{\text{Symbol}}\right) \times \overline{\text{Symbol}}\right)$, which is a set of mappings where each mapping contains a set of attribute partitions from the left-hand side and a set of attribute partitions from the right-hand side to a single partition in the join result.

$$H \backslash P \sqcup^{J}_{M_1,M_2} H' \backslash P' = H'' \backslash P''$$

For each matched partition $(\bar{\text{F}}_1, \bar{\text{F}}_2, \text{F}') \in J$, F' must over-approximate $\bigcup \bar{\text{F}}_1$ and $\bigcup \bar{\text{F}}_2$. Similarly, for each pair $\text{A}_i \mapsto \text{A}'$ in M_1 and M_2, A' in the output of the join must over-approximate A_i in the appropriate input to the join. The algorithm for the join is detailed in [8].

4.1 Desynchronized Separation

Desynchronized terms can be introduced at any function call. They are automatically derived by evaluating all of the arguments to symbols, possibly eliminating already existing desynchronized terms to do so. Once this has been completed, a special function reach is used to determine the desynchronized region.

$$\mathrm{reach} : \mathcal{P}\left(\overline{\mathrm{Symbol}}\right) \times \widehat{\mathrm{Heap}} \to \widehat{\mathrm{Heap}}_u \times \widehat{\mathrm{Heap}}_r$$

The function reach returns a partitioning (H_u, H_r) of the passed heap. The partition H_r is the part possibly reachable from the arguments of the function, including the global object and any closed variables. The partition H_u is the part unreachable from the arguments of the function. With reach, a frame H_u is inferred. The introduction of desynchronization is given with a transfer function judgment and relies on an abstract environment $\hat{E}$ to map variables to abstract addresses and then relates a pre abstract state D_1 to a post abstract state D_2 via a command c:

$$\boxed{\hat{E} \vdash [D_1]\, c\, [D_2]}$$

DESYNC-INTRO

$$\frac{\begin{array}{c}\hat{E}(x) = \mathrm{V}_f \qquad \hat{E}(y_1) = \mathrm{V}_1 \quad \cdots \quad \hat{E}(y_n) = \mathrm{V}_n \\ \mathrm{reach}(\{\mathrm{V}_f, \mathrm{V}_1, ..., \mathrm{V}_n\}, H) = (H_u, H_r) \qquad H' = H_u * [\![H_r]\!]\mathbf{call}\ \mathrm{V}_f(\mathrm{V}_1, ..., \mathrm{V}_n)\end{array}}{\hat{E} \vdash [H \mid P]\ \mathbf{call}\ x(y_1, ..., y_n)\ [H' \mid P]}$$

The splitting of the heap into the function frame H_u and the function footprint H_r is heuristic. For the analysis to be successful on resynchronization, the footprint H_r should over-approximate all memory that could be accessed by any function to which this call could resolve. But then this desynchronized memory H_r is no longer accessible in the analysis of the code after this call (i.e., when accessed, a warning will be raised). Thus, it may be that with an imprecise $\mathrm{reach}()$, the analysis cannot proceed either on the code after the call or on resynchronization. In our implementation, we define reach to yield H_r as the entire reachable heap from the arguments [25], so we allow any function be used for later resynchronization.

Example 3 (Desynchronization introduction). In Figure 5 there is a call to the client-supplied initialization function. This is a function that originated outside the class library and thus is necessarily undefined. When this call occurs, we introduce a desynchronized term representing the effects of this constructor. We use an "arrow-following" $\mathrm{reach}()$ function that determines that one (shown) object is reachable from the arguments and thus in H_r at ⓐ: $\{\mathrm{a}_2\}$. This leaves the objects pointed to by `attrs` in H_u. The resulting introduced desynchronized term is shown in ⓑ.

In other abstract domain operations such as transfer functions, join, widening, or inclusion checking on a domain constructed with desynchronized separation, desynchronized terms must be treated as unknown, but separate portions of the heap. As a consequence desynchronized memory is inaccessible as part of transfer functions and any transfer function that must access it may not proceed:

DESYNC-FRAME

$$\frac{\hat{E} \vdash [H \mid P]\, c\, [H' \mid P]}{\hat{E} \vdash [H * [\![H_d]\!]\mathbf{call}\ \mathrm{V}_f(\mathrm{V}_1, ..., \mathrm{V}_n) \mid P]\, c\, [H' * [\![H_d]\!]\mathbf{call}\ \mathrm{V}_f(\mathrm{V}_1, ..., \mathrm{V}_n) \mid P]}$$

This DESYNC-FRAME rule is a special case of the separation logic frame rule that frames out the desynchronized part of memory and applies the transfer function to the remainder of memory. If this is not well defined because memory in the result of the desynchronized term must be accessed, either a different definition of $\mathrm{reach}()$ should be used or the code must be changed to ensure that the needed memory is not in a desynchronized region.

Similar rules apply for join, widening, and inclusion checking. Desynchronized regions can be joined or widened if they syntactically match, producing the same desynchronized region. Otherwise, without employing a variety of precondition generalization, a join or widening can only be completed if the logic supports TRUE, in which case all precision for this region is lost. Similarly for inclusion checking, only if there is a syntactic match does it return true for desynchronized regions.

Introduction heuristics and elimination: For the purposes of analyzing JavaScript libraries, we use a simple introduction heuristic for desynchronized terms: if a function call can be resolved to a known function, a desynchronized term should not be introduced. This policy has the effect that desynchronized terms only represent unknown functions and thus we do not want to eliminate these terms from the heap. In fact, they nicely represent the callback behavior that occurs in the library in the library's inferred postcondition.

However, there are circumstances where such a simple heuristic may be non-optimal, and it may be desirable to introduce desynchronized terms even when the code for a called function is available. For example, sufficiently surjective functions [28] are functions where after a number of recursions the effect of continued recursion does not matter. In these situations desynchronization can represent the behavior of the unbounded number of recursive calls without actually evaluating all of those calls. Another situation where desynchronization can benefit is in speeding up the analysis when known functions may take too long to analyze but where they do not affect the result in any meaningful way. In these situations, the postcondition includes a desynchronized term that refers to the known function, but the result of that function has not been evaluated.

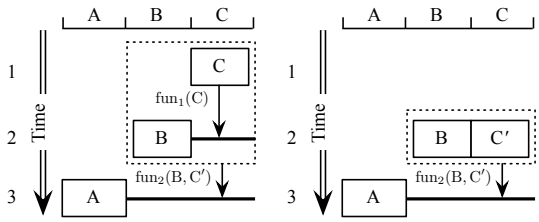
If desynchronized terms are introduced anywhere, it may be necessary that due to access of desynchronized memory, the term that describes that memory has to be eliminated. This can be done if, for example, the synchronizing function's code is available. The resynchronization process takes advantage of the separation logic frame rule by running the analysis on the synchronizing function starting from the desynchronized term:

$$\textsc{Desync-Elim}\quad \frac{\cdot \vdash [H_d \restriction P]\,\mathbf{call}\ \mathrm{V}_f(\mathrm{V}_1,...,\mathrm{V}_n)\,[H'_d \restriction P] \qquad \hat{E} \vdash [H * H'_d \restriction P]\,c\,[H' \restriction P]}{\hat{E} \vdash [H * [\![H_d \restriction P]\!]\mathbf{call}\ \mathrm{V}_f(\mathrm{V}_1,...,\mathrm{V}_n)]\,c\,[H' \restriction P]}$$

With such an elimination rule it is possible to eagerly introduce desynchronized terms on every function call and then lazily eliminate them as portions of the heap are needed.

When employing such an elimination rule, it is possible to consider the variety of ways in which the $\cdot \vdash [H_d \restriction P]\,\mathbf{call}\ \mathrm{V}_f(...)\,[H'_d \restriction P]$ judgment could be satisfied. One way is if each function in V_f can be resolved to known code. In this case the analyzer can be run on each resolvent and a disjunction of postconditions considered. Alternatively, the formula H could carry the information to satisfy this judgment in the form of a nested Hoare triple [26].

Example 4 (Desynchronization elimination). A region of the heap can be resynchronized by eliminating a desynchronized term:



Here, the region C is resynchronized with B by analyzing the call to $\mathrm{fun}_1(\mathrm{C})$ starting from the memory state C resulting in memory state C′. Note that this resynchronization does not require analyzing $\mathrm{fun}_2(\mathrm{B},\mathrm{C}')$. This combined region can stay desynchronized if none of the desynchronized memory is required to proceed with the analysis.

Theorem 1 (Soundness of desynchronization introduction). *Desynchronization introduction is sound because the following property holds: for all* $E, \hat{E}, \sigma, \sigma', H, H', P$. $E \vdash \langle\sigma\rangle \mathbf{call}\ x(y_1,...,y_n)\langle\sigma'\rangle$ *and* $\hat{E} \vdash [H \natural P]\ \mathbf{call}\ x(y_1,...,y_n)\ [H' \natural P]$ *and* $(\eta,\sigma) \in \gamma(H \natural P)$ *implies that there exists* η' *such that* $(\eta',\sigma') \in \gamma(H' \natural P)$.

4.2 Attribute/Value Trackers

The primary benefit of attribute/value trackers occurs when they can be preserved from one abstract state to the next. To do so requires extending HOO transfer functions for the multi-state abstractions. The extension is trivial by appending the abstract heap from the precondition to each state in the transfer functions:

$$\frac{\hat{E} \vdash [H \natural P]\, c\, [H' \natural P]}{\hat{E} \vdash \left[[H]_{H_1} \natural P\right] c \left[[H']_{H_1} \natural P\right]}$$

To utilize attribute/value trackers, they must be introduced and managed appropriately. The goal is to reuse the same tracker whenever it is possible to do so and to only introduce fresh trackers when it is otherwise impossible. A key aspect of trackers is that the domain of a tracker is determined by the corresponding attribute set F at the point of introduction and thus the same tracker can be applied to any attribute set F′ such that $\mathrm{F}' \subseteq \mathrm{F}$ if the values also match appropriately.

There are three key steps in managing this behavior of attribute/value trackers. First, materialization is responsible for splitting a singleton set off of a summary. In doing so, trackers can be preserved, even when the partition tied to a particular tracker is split. Second, trackers can be transfered along with attributes and values when an attribute/value pair is copied from one object to another. Finally, trackers can be introduced when not otherwise available.

Materializing with attribute/value trackers: Since JavaScript does not have operations that allow many attributes and values to be copied or manipulated at once, a key operation for maintaining precision with attribute/value trackers is preserving them when splitting summarized objects/attributes/values so that there is a single attribute/value pair from a single object to be copied to another object. This operation is *materialization* and is described in Figure 8 in three parts.

Each materialization rule is of the form $D_1 \Rightarrow D_2$ and thus intended to be used with the rule of consequence from Hoare logic [16] to allow a future rule to be applied. For example,

$$\boxed{D_1 \Rightarrow D_2}$$

MAT-VALUE

$$\frac{\mathrm{v}\text{ is fresh} \qquad P' = P \wedge \{\mathrm{v}\} \subseteq \mathrm{V}}{[H_2 * \{\mathrm{a}\} \cdot \langle O; \{\mathrm{f}\} : \rho \mapsto \mathrm{V}\rangle]_{H_1} \restriction P \Rightarrow [H_2 * \{\mathrm{a}\} \cdot \langle O; \{\mathrm{f}\} : \rho \mapsto \{\mathrm{v}\}\rangle]_{H_1} \restriction P'}$$

MAT-ATTR

$$\frac{\mathrm{F}'\text{ is fresh} \qquad P' = P \wedge \{\mathrm{f}\} \uplus \mathrm{F}' = \mathrm{F} \qquad P'' = P \wedge \{\mathrm{f}\} \cap \mathrm{F} = \emptyset}{\begin{array}{l}[H_2 * \{\mathrm{a}\} \cdot \langle O; \mathrm{F} : \rho \mapsto \mathrm{V}\rangle]_{H_1} \restriction P \Rightarrow \\ \quad [H_2 * \{\mathrm{a}\} \cdot \langle O; \mathrm{F}' : \rho \mapsto \mathrm{V}; \{\mathrm{f}\} : \rho \mapsto \mathrm{V}\rangle]_{H_1} \restriction P' \vee [H_2 * \{\mathrm{a}\} \cdot \langle O; \mathrm{F} : \rho \mapsto \mathrm{V}\rangle]_{H_1} \restriction P''\end{array}}$$

MAT-ADDR

$$\frac{\mathrm{A}'\text{ is fresh} \qquad P' = P \wedge \{\mathrm{a}\} \uplus \mathrm{A}' = \mathrm{A} \qquad P'' = P \wedge \{\mathrm{a}\} \cap \mathrm{A} = \emptyset}{[H_2 * \mathrm{A} \cdot \langle O\rangle]_{H_1} \restriction P \Rightarrow [H_2 * \{\mathrm{a}\} \cdot \langle O\rangle * \mathrm{A}' \cdot \langle O\rangle]_{H_1} \restriction P' \vee [H_2 * \mathrm{A} \cdot \langle O\rangle]_{H_1} \restriction P''}$$

Fig. 8: Materialization of all of the parts of objects never produces fresh attribute/value trackers. It reuses existing trackers.

rules for assignment (next section) can only be applied to singleton object addresses, singleton attributes, and often singleton values. By applying materialization correctly, an abstract heap element that consists of summary object addresses, summary attributes, and summary values can be converted to the appropriate singleton form without loss of precision, assuming a precise pure domain.

The first rule for materialization MAT-VALUE materializes a single value from a summary value, assuming that the object address, and attribute are already materialized. Because the object address and attribute are singletons, it must be that there is a singleton value $\{\mathrm{v}\}$ and thus it can be materialized from the summary V. Doing so produces the additional constraint that $\{\mathrm{v}\}$ is a subset of V. Because the materialized value $\{\mathrm{v}\}$ is a fresh variable, this added constraint does not affect soundness.

The second rule for materialization is the primary rule for materializing attribute/value trackers. The MAT-ATTR rule splits an attribute set F into two attribute sets F′ and $\{\mathrm{f}\}$. There are two possible outcomes of this split. Either $\{\mathrm{f}\}$ was already a subset of F, in which case the materialization can proceed, or $\{\mathrm{f}\}$ is disjoint from F, in which case there is no materialization. In the case that the materialization proceeds, when the set F is split into two, both new partitions can be assigned the same tracker as was present in the original partition. This is because such a split does not require an extension of the domain of the tracker.

This second rule is applied whenever an object is being read. The attribute that is being read must be materialized from each partition of the object that may contain the attribute in question. Therefore, the read operation must consider a case where the attribute is in each partition of the object. The resulting pure constraints of MAT-ATTR may thus produce conflicts, causing such cases to be dropped.

The third rule for materialization MAT-ADDR also manipulates attribute/value trackers, but less directly than the previous rule. This rule materializes a particular address $\{\mathrm{a}\}$ from

A-OVERWRITE-DISTINCT

$$\frac{\hat{E}(x_1)=\mathrm{a}_1 \qquad \hat{E}(y)=\mathrm{f} \qquad \hat{E}(x_2)=\mathrm{a}_2}{\hat{E}\vdash \begin{array}{c}\left[\left[H_1{*}\{\mathrm{a}_1\}\cdot\langle O_1;\{\mathrm{f}\}:\rho_1\mapsto \mathrm{V}_1\rangle{*}\{\mathrm{a}_2\}\cdot\langle O_2;\{\mathrm{f}\}:\rho_2\mapsto\{\mathrm{v}_2\}\rangle\right]_{H_0}{}^{\shortmid}P\right]\\ x_1[y]:=x_2[y]\\ \left[\left[H_1{*}\{\mathrm{a}_1\}\cdot\langle O_1;\{\mathrm{f}\}:\rho_2\mapsto\{\mathrm{v}_2\}\rangle{*}\{\mathrm{a}_2\}\cdot\langle O_2;\{\mathrm{f}\}:\rho_2\mapsto\{\mathrm{v}_2\}\rangle\right]_{H_0}{}^{\shortmid}P\right]\end{array}}$$

Fig. 9: Example abstract transfer function for assignment where the attribute/value tracker ρ_2 is transfered from the object at a_2 to the object at a_1

a summary of addresses A. Like the previous rule, if $\{\mathrm{a}\}$ is a subset of A, the summary can be split. When this split occurs, the whole object definition is duplicated. Consequently each tracker is also duplicated. In the event that the materialization cannot occur, this constraint is added to indicate in the future that such an attempt was already considered.

Example 5 (Materializing a summary). Consider the heap abstraction $[\mathrm{A}\cdot\langle\mathrm{F}:\rho\mapsto\mathrm{V}\rangle]_{H_1}{}^{\shortmid}\{\mathrm{a}\}\subseteq\mathrm{A}\wedge\{\mathrm{f}\}\subseteq\mathrm{F}$. If the analysis needs to read from $\mathrm{a}[\mathrm{f}]$, this must be materialized. To achieve the following heap abstraction first the MAT-ADDR rule is applied, then the MAT-ATTR rule is applied to the result, then the rule MAT-VALUE is applied:

$$\left[\begin{array}{l}\mathrm{A}'\cdot\langle\mathrm{F}:\rho\mapsto\mathrm{V}\rangle{*}\\ \{\mathrm{a}\}\cdot\langle\mathrm{F}':\rho\mapsto\mathrm{V};\{\mathrm{f}\}:\rho\mapsto\{\mathrm{v}\}\rangle\end{array}\right]_{H_1}{}^{\begin{array}{c}\{\mathrm{a}\}\uplus\mathrm{A}'=\mathrm{A}\\ \shortmid\wedge\{\mathrm{f}\}\uplus\mathrm{F}'=\mathrm{F}\\ \wedge\{\mathrm{v}\}\subseteq\mathrm{V}\end{array}}$$

Transfering attribute/value trackers: Attribute/value trackers are transfered from one object to another by assignment. For simplicity, we assume here that all assignments between objects are transformed into the form of a simultaneous read from an object and a write to another object. When the attribute being read and written matches so that an attribute/value pair is being copied, there is an opportunity to transfer that attribute/value pair from one object to the other. When this transfer happens, the attribute/value tracker can be transfered as well.

Figure 9 shows one of the transfer functions that enables an attribute/value tracker transfer. The A-OVERWRITE-DISTINCT rule uses the abstract environment $\hat{E}$ to map variables onto addresses and then if the same attribute exists in two distinct objects the transfer occurs, in this case replacing ρ_1 with ρ_2.

Introducing attribute/value trackers: Attribute/value trackers should be introduced at chosen program points where the first of the paired states is selected. For example, when constructing an initial abstract state, it would be reasonable to express it as $[H]_H{}^{\shortmid}P$ where the two described heaps are identical. In this instance, fresh attribute/value trackers should be introduced for each partition in H. This establishes the initial relationship between the initial abstract state and the current abstract state and then any attribute/value trackers that are preserved strengthen the relationship between the two states.

Additionally, attribute/value trackers can be introduced at other times. The benefits of doing so are less significant as freshly introduced trackers cannot relate objects from one time to another, but instead are limited to relating multiple objects in the same time. However as track-

ers are incomparable unless they are equal, freely introducing fresh trackers will prevent inclusion checking from succeeding and prevent the analysis from terminating. In the current implementation, we avoid this problem by only introducing absent trackers – after the precondition.
Other domain operations: Other domain operations such as join, widening, and inclusion check are largely the same as with HOO. Attribute/value trackers form a partition-by-partition lattice where any tracker $\rho \sqsubseteq -$. Join, widening, and inclusion follow from this: identical trackers can be matched and maintained through join and widening. Differing trackers must be replaced with –.

Theorem 2 (Soundness of tracker materialization). *Tracker materialization is sound because the following property holds:*
For all $D,D',\eta,\sigma_1,\sigma_2$. $D \Rightarrow D'$ *and* $(\eta,\sigma_1,\sigma_2) \in \gamma(D)$ *implies that* $(\eta,\sigma_1,\sigma_2) \in \gamma(D')$.

Theorem 3 (Soundness of transfer functions). *Transfer functions including desynchronization introduction, elimination, framing, and attribute/value tracker transfer are sound because the following property holds:*
For all $D,D',\sigma,\sigma',\sigma_0,\eta$. $E \vdash \langle\sigma\rangle c \langle\sigma'\rangle$ *and* $\hat{E} \vdash [D]\, c\, [D']$ *and* $(\eta,\sigma_0,\sigma) \in \gamma(D)$ *implies that there exists a* η' *such that* $(\eta',\sigma_0,\sigma') \in \gamma(D')$

5 Empirical Evaluation

In this section, we evaluate the use of desynchronized separation and attribute/value trackers on JavaScript meta-feature libraries – libraries that add language features to JavaScript through the use of object manipulation and callbacks. To do so, we test two hypotheses: (1) Does desynchronization provide the necessary precision for analyzing libraries that call unknown functions. (2) Do attribute/value trackers provide necessary precision for analyzing libraries that manipulate objects with unknown attribute/value relationships.

To evaluate these hypotheses, we identified several classes of meta-feature libraries that are available in JavaScript: classes, traits[3], mixins[4], and memoization[5]. From each of these candidates, we selected a small, but complex core (Table 1a) and annotated that functionality with preconditions. These preconditions indicate aliasing in the heap as well as give names to sets of attributes. Then, on each library, we compared expected postconditions against those generated by the JSAna analyzer for JavaScript, which is based on HOO with desynchronized separation and attribute value trackers.

The results of these experiments are shown in Table 1b. The first two properties are able to be proven solely with HOO. In the Traits example, which combines two objects into one, when the same attribute is present in both source objects, a single, global conflict value is used in the place of either source value. Because it is a single value, partitioning is sufficient to distinguish it. Similarly, in Memo, while Memo makes a call to an unknown function, if the precondition indicates that the call has already been memoized, that function call never happens and thus HOO's object-level reasoning, given a sufficiently precise set domain, is fully precise.

The second two properties actually require analyzing calls to unknown functions. In Class, this is the call to the initializer, and in Memo, this is the call to the memoized function. In

[3] Extracted from http://soft.vub.ac.be/~tvcutsem/traitsjs/ [4] Extracted from http://prototypejs.org/
[5] Extracted from https://developers.google.com/closure/library/

Table 1: Results of running HOO with desynchronized separation and attribute/value trackers on JavaScript meta-feature libraries.

(a) Test Library Code: Stmts is the number of statements in the program after preprocessing and lowering. Vars is the peak number of pure symbols used in the analysis. JP is the number of join points.

Test	Stmts	Vars	JP	Time (s)
Mixin	33	52	1	0.16
Traits	131	111	1	7.20
Memo	149	179	0	0.24
Class	128	118	1	8.13

(b) Properties: HOO is a property proven solely by HOO. D is HOO with desynchronized separation. T is HOO with attribute/value trackers. D+T is HOO with both enhancements.

Test	Property	HOO	D	T	D+T
Traits	Conflict managed	✓	✓	✓	✓
Memo	In table	✓	✓	✓	✓
Class	Constructor Call	✗	✓	✗	✓
Memo	Call saved	✗	✓	✗	✓
Mixin	Object extended	✗	✗	✓	✓
Traits	Object extended	✗	✗	✓	✓
Class	Resulting Object	✗	✗	✗	✓

both cases, the reachability analysis identifies suitable heap regions to allow the analysis to be fully precise. By comparison HOO, without desynchronization, cannot handle these calls and thus cannot prove the desired property.

The two object extended properties reason about the precise extension of objects that occurs in mixins and traits. In Mixin, an existing object has a number of attributes and corresponding values that may be overwritten by adding attributes and values from another object into it. Similarly, the Traits adds attribute and values from two different objects. Maintaining exact relationships between attributes and values is impossible without the use of attribute/-value trackers, which allow the inferred postconditions for these analyses to be fully precise.

The last property, which checks that the instance created by the class is correct requires both attribute/value trackers and desynchronization to be precise. Because it uses both object manipulations and calls to the initializer, this indicates that these two additions are complementary and necessary for analyzing meta-feature libraries in JavaScript.

While it is not a goal to highly optimize for performance at this time, the results suggest that the analysis time is dependent on the number of pure symbols (Vars) and the number of join points (JP). When the number of variables increases (as long as there are join points), the overall analysis slows down. As in [8], nearly all of the cost can be attributed to the exponential set domain, which is implemented using binary decision diagrams. On top of this, the overhead of adding desynchronization and attribute/value trackers is negligible.

6 Discussion

In this section, we discuss the features and limitations of the analysis by considering two of the benchmarks in more detail. Additionally, we give some perspective on situations where the analysis loses precision.

6.1 Case Study: Class

The class benchmark is similar to the function `Class` presented in the introduction and the overview. Here we examine the similarities between the theory presented in the overview and what occurs in practice. We use program points from the overview for reference to the code used in the benchmark (which is complicated by more complete JavaScript support).

The analysis of the `copy` function proceeded exactly as shown in the overview. On each iteration of the analysis, a tracker was duplicated via materialization. That tracker was transfered to the `result` object. Consequently, the postcondition ② of copy was fully precise.

The desynchronization also works as expected. Critically, reachability identifies that a_1 and the local variable `result` are both *outside* the desynchronized region. This means that these things are unmodified by the call to the client-supplied initializer. Consequently, the resulting postcondition shows that the result object is the object created by the constructor and that constructor always produces exactly the same object attributes *and values* prior to the call to the client-supplied initializer regardless of how many times it is called.

6.2 Case Study: Memoization

The Memo benchmark transforms a function into a memoized version of that function. To accomplish this, it first translates the arguments array into a unique identifier by calling a $\mathrm{uid}()$ function passing it the entire arguments object. Then it determines if that unique identifier is already in the memoization table. If so, it returns the value from the table. Otherwise it calls the function to be memoized, f, passing it arguments (via JavaScript's apply functionality) and then memoizing the result.

Each of the function calls is challenging. The $\mathrm{uid}()$ function is essentially a hash function. It is responsible for converting data of any type into a unique string suitable for use in indexing into an object. Because hash functions are typically hard to analyze and this is a hash function that hashes to strings, this function presents a problem for analysis. Even if we had the code for it, it would be undesirable to analyze it.

The second function call is also challenging because it is a callback into client-supplied code. The behavior of the function could be anything. It could have side-effects or it could be pure. Its only restriction is from JavaScript being memory safe (it cannot create pointers to previously unreachable parts of the heap).

Both of these problems are addressed by desynchronization as shown in Figure 10. Figure 10 shows as representation of the postcondition of the library function. In it we can see that not only was the callback to the client-supplied function $\mathrm{f}()$ desynchronized, but the call to $\mathrm{uid}()$ was desynchronized. Additionally, because the arguments object may have been modified by the $\mathrm{uid}()$ function, it is necessary to nest the desynchonizations to represent the result.

Nested desynchronization allows continuation-like behavior to be analyzed over parts of the program. Here the arguments object was possibly modified by the uid function before being possibly modified by the callback. The benefit of this nested structure is even if there is a sequence of functions that all touch the same memory, analysis can proceed by nesting all of these individual functions.

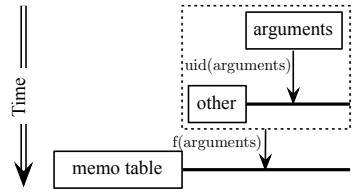


Fig. 10: Desynchronization phases of the memoization example

6.3 Boundaries of Analysis and Future Improvements

While our results suggest that both desynchronization and attribute/value trackers can be effective on JavaScript code, there are limitations to the precision. The most significant limitation is that attribute/value trackers are dropped when direct copies are not used. In particular, complex, nested copies are not currently supported by these trackers. For example, the following code wraps each value inside a newly allocated object.

```
result[a] = {value: attrs[a]};
```

Without the ability to reason about intermediary objects, full precision cannot be maintained and such abstractions fall back to what HOO can do. However, this behavior does not appear to occur in most libraries and thus may not be a significant issue. Adding support for this particular case is another form of tracker, but the inference of such trackers remains challenging.

While we find that reachability is a suitable heuristic for the analysis of many libraries, it may be overly pessimistic. In certain situations developers intentionally make portions of libraries globally mutable, but mutation is still not the common case.

7 Related Work

JavaScript specification and analysis: JuS [12] is an abduction-based inference tool for JavaScript targeted at the prototype and the scope chain. It is based on a detailed model of JavaScript semantics [2, 11] and thus automation is limited to resolving variable lookup through a prototype and scope chain. DJS [5, 6], which is a specification language and a dependent refinement type system for JavaScript, by comparison is more restricted in its support of JavaScript and thus offers more automation in that straight-line code can be reliably analyzed (loops and functions require annotations). The work presented in this paper automates discovery of loop invariants and callback summaries. This is significantly more automation than is provided by existing systems without sacrificing language features.

TAJS [17, 18], WALA [27], JSAI [15, 19], and SAFE [1, 21] are whole-program JavaScript analyses. Unlike the above systems, they require no annotations at all and are highly automated. However, they are ill suited to analyzing partial programs as is the case when verifying libraries. Because whole-program analysis has extensive context information, including object attributes and function bodies, there is less complexity involved in handling first-class functions (the function body can usually be resolved) or open objects (the attribute names are often fully known) and thus the abstractions used by these analyzers are incomparable to those that we employ.

The idea of attribute/value trackers comes from correlation tracking [27], which is implemented in both WALA and TAJS. Correlation tracking uses context sensitivity to exactly determine the constant attribute/symbolic value pairs needed for loops. Attribute/value trackers generalize this to symbolic attribute/symbolic value pairs that are each elements of summaries.

Higher-order separation logic and contracts: Desynchronized separation is closely tied to the concept of nested Hoare triples [26] and higher-order separation logic [20]. However, there are several key differences.

The goal of desynchronized separation is fundamentally different from that of nested Hoare triples. Unlike desynchronized separation, nested Hoare triples are intended to be used in program logics and not for automated inference. While there are efforts to automate some amount of reasoning [4], current techniques require significant annotation overhead and perform no inference, only inclusion checking.

The other significant difference is that nested Hoare triples strive for complete generality. A desynchronized term carries the following correspondence with nested Hoare triples:

$$[\![H_1]\!]\mathbf{call}\ \mathrm{V}_f(\mathrm{V}_1,...\mathrm{V}_n)*H_o \qquad \Rightarrow \qquad \exists H_2.\,[H_1]\,\mathbf{call}\ \mathrm{V}_f(\mathrm{V}_1,...\mathrm{V}_n)\,[H_2]\wedge H_2*H_o$$

where an equivalence holds if an appropriate H_2 is chosen. The additional heap H_o is here to illustrate the key differentiating factor. A nested Hoare triple is a pure part of a formula that describes a value whereas a desynchronized term describes a heap that results from calling a function. The $*H_o$ illustrates which parts of the description are heap and which are pure.

The process of inference using desynchronization is significantly simpler than using nested Hoare tripes. This is due to the fact that desynchronization is less expressive than nested Hoare triples. There are fewer existentially quantified variables, and there is no need to treat portions of the heap that are simply passed through the unknown function call as separate portions of the heap that are manipulated. As a result, it is possible to (1) easily adapt existing separation-logic-based analyses to certain higher-order tasks and (2) easily perform necessary heap splits during the analysis because there are two possible ways the heap can be split.

The key idea of nested Hoare triples is also similar to static contract checking for higher order languages [23, 29], which requires a pure specification of any callback's behavior up front. It is also similar to [22], except that it relies on separation logic and is applied to a stronger heap abstraction.

The goal of desynchronized separation is to not require a specification for callbacks at all, if the developer is judicious with built-in language protection mechanisms. In the event that memory is insufficiently protected, or the reachability analysis is too coarse, our analysis could be extended with nested Hoare triple specifications. In such a scenario, the nested Hoare triple is essentially the same as a resolvable function call. However, it is possible to imagine a simpler specification where only a footprint for the unknown function is specified. In this case, desynchronization would be required, but it would be applied to specified footprint (instead of using heap reachability to determine the split).

Container analysis: A significant part of HOO resembles an analysis for containers. Keeping track of object attribute names and values is similar to what is required for reasoning about mapping containers. The analysis in [10] also uses uninterpreted functions. However, the purpose of their uninterpreted functions is not to keep track of unknown attribute/value relationships, but instead to handle the sparsity problem of containers. Instead, they use uninterpreted functions to map a elements of a key/attribute type to a natural number that is

the array index containing the value. The value arrays are then represented and manipulated using fluid updates [9].

Uninterpreted functions: There are several analyses [3, 13] that use uninterpreted functions to combine multiple abstract domains. While this work is also used for object and heap abstractions, the purpose of uninterpreted functions is different from attribute/value trackers. The uninterpreted functions in [3, 13] are used to transfer information between multiple abstract domains, whereas attribute/value trackers disambiguate individual symbolic elements of summaries across an analysis.

8 Conclusion

In this paper, we presented two multi-state abstractions that build upon abstract domains for heaps like HOO. Desynchronized separation gives a means for automatically reasoning about callbacks to unknown functions, while attribute/value trackers improve upon the partitioning of object attributes performed by HOO by maintaining consistent relationships between symbolic attribute names and symbolic values that are both members of summaries. Collectively these multi-state abstractions enable precise analysis of several core routines in JavaScript libraries.

Acknowledgements. Thank you to Anders Møller, our anonymous reviewers, and members of CUPLV and Antique for the helpful reviews and feedback. This material is based upon work supported in part by a Chateaubriand Fellowship, by the National Science Foundation under Grant Numbers CCF-1055066 and CCF-1218208, and by the European Research Council under the FP7 grant agreement 278673 (Project MemCAD).

References

[1] SungGyeong Bae, Hyunghun Cho, Inho Lim, and Sukyoung Ryu. $\text{SAFE}_{\text{WAPI}}$: Web API misuse detector for web applications. In *FSE*, 2014.

[2] Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. A trusted mechanised JavaScript specification. In *POPL*, pages 87–100, 2014.

[3] Bor-Yuh Evan Chang and K. Rustan M. Leino. Abstract interpretation with alien expressions and heap structures. In *VMCAI*, pages 147–163, 2005.

[4] Nathaniel Charlton, Ben Horsfall, and Bernhard Reus. Crowfoot: A verifier for higher-order store programs. In *VMCAI*, pages 136–151, 2012.

[5] Ravi Chugh, David Herman, and Ranjit Jhala. Dependent types for JavaScript. In *OOPSLA*, pages 587–606, 2012.

[6] Ravi Chugh, Patrick Maxim Rondon, and Ranjit Jhala. Nested refinements: a logic for duck typing. In *POPL*, pages 231–244, 2012.

[7] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[8] Arlen Cox, Bor-Yuh Evan Chang, and Xavier Rival. Automatic analysis of open objects in dynamic language programs. In *SAS*, 2014.

[9] Isil Dillig, Thomas Dillig, and Alex Aiken. Fluid updates: Beyond strong vs. weak updates. In *ESOP*, pages 246–266, 2010.

[10] Isil Dillig, Thomas Dillig, and Alex Aiken. Precise reasoning for programs using containers. In *POPL*, pages 187–200, 2011.
[11] Philippa Gardner, Sergio Maffeis, and Gareth David Smith. Towards a program logic for JavaScript. In *POPL*, pages 31–44, 2012.
[12] Philippa Gardner, Daiva Naudziuniene, and Gareth Smith. JuS: Squeezing the sense out of JavaScript programs. In *JSTools*, 2013.
[13] Sumit Gulwani and Ashish Tiwari. Combining abstract interpreters. In *PLDI*, pages 376–386, 2006.
[14] Nicolas Halbwachs and Mathias Péron. Discovering properties about arrays in simple programs. In *PLDI*, pages 339–348, 2008.
[15] Ben Hardekopf, Ben Wiedermann, Berkeley R. Churchill, and Vineeth Kashyap. Widening for control-flow. In *VMCAI*, pages 472–491, 2014.
[16] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.
[17] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for JavaScript. In *SAS*, pages 238–255, 2009.
[18] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Interprocedural analysis with lazy propagation. In *SAS*, pages 320–339, 2010.
[19] Vineeth Kashyap, John Sarracino, John Wagner, Ben Wiedermann, and Ben Hardekopf. Type refinement for static analysis of JavaScript. In *DLS*, pages 17–26, 2013.
[20] Neelakantan R. Krishnawami. *Verifying Higher-Order Imperative Programs with HIgher-Order Separation Logic*. PhD thesis, Carnegie Mellon University, 2011.
[21] Hongki Lee, Sooncheol Won, Joonho Jin, Junhee Cho, and Sukyoung Ryu. SAFE: Formal specification and implementation of a scalable analysis framework for ECMAScript. In *FOOL*, 2012.
[22] Ravichandhran Madhavan, G. Ramalingam, and Kapil Vaswani. Modular heap analysis for higher-order programs. In *SAS*, pages 370–387, 2012.
[23] Phuc C. Nguyen, Sam Tobin-Hochstadt, and David Van Horn. Soft contract verification. In *ICFP*, 2014.
[24] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74, 2002.
[25] Noam Rinetzky, Mooly Sagiv, and Eran Yahav. Interprocedural shape analysis for cutpoint-free programs. In *SAS*, pages 284–302, 2005.
[26] Jan Schwinghammer, Lars Birkedal, Bernhard Reus, and Hongseok Yang. Nested Hoare triples and frame rules for higher-order store. *Logical Methods in Computer Science*, 7(3), 2011.
[27] Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip. Correlation tracking for points-to analysis of JavaScript. In *ECOOP*, pages 435–458, 2012.
[28] Philippe Suter, Ali Sinan Köksal, and Viktor Kuncak. Satisfiability modulo recursive programs. In *SAS*, pages 298–315, 2011.
[29] Dana N. Xu, Simon L. Peyton Jones, and Koen Claessen. Static contract checking for Haskell. In *POPL*, pages 41–52, 2009.