# Automatic Analysis of Open Objects in Dynamic Language Programs

Arlen Cox[1], 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.** In dynamic languages, objects are *open*—they support iteration over and dynamic addition/deletion of their attributes. Open objects, because they have an unbounded number of attributes, are difficult to abstract without a priori knowledge of all or nearly all of the attributes and thus pose a significant challenge for precise static analysis. To address this challenge, we present the HOO (Heap with Open Objects) abstraction that can precisely represent and infer properties about open-object-manipulating programs without any knowledge of specific attributes. It achieves this by building upon a relational abstract domain for sets that is used to reason about partitions of object attributes. An implementation of the resulting static analysis is used to verify specifications for dynamic language framework code that makes extensive use of open objects, thus demonstrating the effectiveness of this approach.

## 1 Introduction

```
for(var p in s)
  if(p in c) r[p] = "conflict";
  else       r[p] = s[p];
```

**Fig. 1** – The essence of open object-manipulating routines.

Static analysis of dynamic languages is challenging because objects in these languages are open. *Open objects* have mutable and iterable attributes (also called fields, properties, instance variables, etc.); developers can programmatically add, remove, and modify attributes of existing objects. Because of their flexibility, open objects enable dynamic language developers to create frameworks with object-manipulating routines [30] that decrease code size, increase code reuse, and improve program flexibility and extensibility. In Fig. 1, we show JavaScript code that conditionally adds attributes to the object r with attributes from object s—code similar to this snippet is repeated in various forms in, for instance, frameworks that implement class and trait systems. Because specific attributes of the objects r, s, and c are unknown, we cannot conclude exactly what the structure of the object r is at the end of this code. However, it can be derived from the structure of the original r, s, and c that each attribute (written $\hat{f}$) in the set of all attributes of r (written attr(r)) can fall into one of three parts. First, if $\hat{f}$ is in both attr(s) and attr(c), the corresponding value is 'conflict'. Second, if $\hat{f}$ is in attr(s) but not in attr(c), the corresponding value is from s. Lastly, if $\hat{f}$ is not in attr(s), the value of attribute $\hat{f}$ of object r is unchanged. In this paper, we argue that inferring these partitions is a solution to what we call the open object abstraction problem.

The *open object abstraction problem* occurs when the attributes of objects cannot be known statically. Unfortunately, the open object abstraction problem significantly

SAS 2014.

increases the difficulty of static analysis. Objects no longer have a fixed set of attributes but instead an unbounded number of attributes. Thus, abstractions of objects must not only abstract the values to which the attributes point but also the attributes themselves. Such abstractions must potentially conflate many attributes into a single abstract attribute. As we demonstrate in this paper, the open object abstraction problem precludes simple adaptations of abstractions for closed-object languages like Java to dynamic languages.

This paper develops the HOO (Heap with Open Objects) abstract domain [7] that does not require knowledge of specific attributes to be precise. It partitions attributes of objects into sets of attributes. Then it relates those sets of attributes with sets of attributes from other objects. Thus, it can represent complex relationships like those that form in the aforementioned example through a relational abstraction for sets. For example, it can automatically infer the three partitions in the attributes of object `r` in the previous example.

Unlike existing analyses that adapt closed-object abstractions [21, 31], the HOO abstract domain is particularly suited for analyzing programs where significant pieces of the program are unknown and thus many attributes of objects are unknown. Because HOO partitions attributes on the fly and relates partitions to one another, it maintains useful information even when unknown attributes are accessed and manipulated. Such information is necessarily lost in closed-object adaptations and thus a domain like HOO is a fundamental building block towards modular analysis of dynamic language programs.

In this paper we make the following contributions:

- We introduce HOO, an abstraction for objects that relates partitions of attributes between multiple objects by building on a relational abstract domain for sets. Using these relations, we directly abstract open objects instead of adapting existing object abstractions that require knowledge of specific attributes. (Section 3).
- We introduce attribute materialization, an operation that extracts individual symbolic attributes from attribute summaries, allowing strong updates of open objects. Using attribute materialization, we derive transfer functions that use strong updates for precisely reading from objects and writing to objects (Section 4).
- We develop algorithms for widening and inclusion checking that are used to automatically infer loop invariants in open-object-manipulating programs. These algorithms use iteration-progress sets to allow strong updates across loop iterations, thus inferring partitions of object attributes (Section 5).
- We evaluate HOO by using inferred post-conditions for object-transforming functions like those commonly found in JavaScript libraries to prove properties about the structure of objects (Section 6).

## 2 Overview

In this section we demonstrate the features of the abstraction by analyzing the example loop from the introduction. Fig. 2 shows key analysis states in the final iteration of abstract interpretation after starting from an annotated pre-condition shown at ①. In this iteration, the analysis proves that the loop invariant is inductive.

Before executing the loop, ① is the abstract state, where we show three separate abstract objects at addresses $\hat{a}_1$, $\hat{a}_3$, and $\hat{a}_5$ (where $\hat{a}_n$ represents a singleton set of addresses and $\hat{A}_n$ represents a summary of addresses) that are pointed to by variables `r`, `s`, and c (shown in dotted circles) respectively. The attributes of r, attr(r) are $\hat{F}_{\mathtt{r}}$ (where $\hat{f}_n$ represents a singleton set of attributes; $\hat{F}_n$ represents any summary of attributes). Similarly, attr(s) is $\hat{F}_{\mathsf{in}} \uplus \hat{F}_{\mathsf{out}}$. Each attribute in attr(r) contains an object address from the summary $\hat{A}_2$ (shown with a double circle). Since many dynamic languages permit reading attributes that do not exist, the partition noti maps to the value of all attributes *not in* the object. If this partition does not exist, the object is *incomplete* and behaves similarly to a C# or a Java object (Section 3). Boxed on the right are constraints on attribute partitions. These constraints are represented by a relational abstraction for sets, such as QUIC graphs [10].

Appropriate partitioning of objects is vital for performing strong updates. To take advantage of strong updates across loop iterations, ① shows a special partitioning of s. The partition $\hat{F}_{\mathsf{in}}$ is the set of all attributes that have not yet been visited by the loop, whereas the partition $\hat{F}_{\mathsf{out}}$ is the set of all attributes that have already been visited by the loop and thus is initially empty. On each iteration an element is removed from $\hat{F}_{\mathsf{in}}$ and placed into $\hat{F}_{\mathsf{out}}$, allowing relationships to represent not just the initial iteration of the loop, but any iteration. We see these relationships in the loop invariant ⓘ.

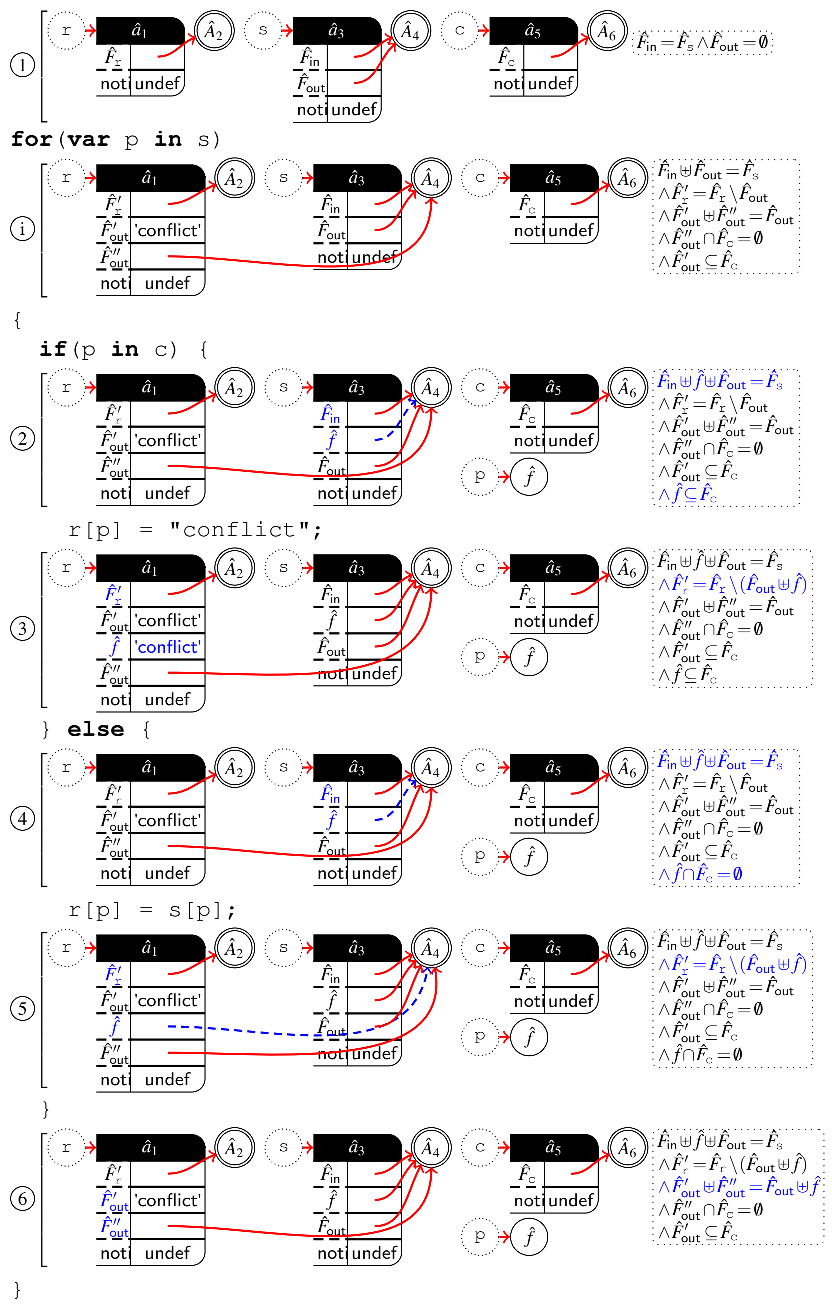
The loop invariant ⓘ shows the three partitions of attr(r) mentioned in the introduction. The partitions are constrained by $\hat{F}_{\mathsf{out}}$, because the overwritten portion of attr(r) can only be from the elements that have already been visited by the loop. Additionally the $\hat{F}''_{\mathsf{out}}$ partition is restricted to have no elements in common with $\hat{F}_{\mathtt{c}}$. This corresponds to the branch within the loop that determines whether 'conflict' or s[p] is written. This invariant was inferred using abstract interpretation [7] by the HOO abstract domain.

Once in the body of the loop, the variable p is bound to a singleton set $\hat{f}$ that is split from $\hat{F}_{\mathsf{in}}$. Depending on the value of $\hat{f}$, one of two cases occurs. In ② we highlight the changes using blue and dashed points-to arrows, showing that $\hat{f}$ is contained in the properties of $\hat{a}_5$ $\hat{F}_{\mathtt{c}}$. Storing 'conflict' into r[p] gives ③ by first removing $\hat{f}$ from all partitions that make up attr(r) and then adding a new partition $\hat{f}$ and thus performing *attribute materialization* of $\hat{f}$ from the object summary. Because $\hat{f}$ is now materialized, subsequent updates to $\hat{f}$ will update the same $\hat{f}$, rather than weakening the value abstraction that corresponds to one of the larger partitions. Here, the abstract value that corresponds to $\hat{f}$ is set to 'conflict'.

The second case writes s[p] to r[p] when $\hat{f}$ is not contained in $\hat{F}_{\mathtt{c}}$. The starting state ④ is like ② except that $\hat{f} \not\subseteq \hat{F}_{\mathtt{c}}$. The result similarly materializes $\hat{f}$ in $\hat{a}_1$ before pointing that partition to the abstract value $\hat{A}_4$. Thus in both branches of the **if**, we perform strong updates in the abstraction. Transfer functions and strong updates are detailed in Section 4.

After the **if**, we join the two abstract states ③ and ⑤. In essence, the join process (Section 5) merges partitions that have common properties. Here, $\hat{f}$ is summarized into $\hat{F}'_{\mathsf{out}}$ in ③ and $\hat{F}''_{\mathsf{out}}$ in ⑤. The three partitions of attr(r) thus arise from the part of attr(r) that was left after materializing $\hat{f}$, and the two branches of the **if**, which is represented in the set domain with a partial path condition. Once $\hat{f}$ is summarized into $\hat{F}'_{\mathsf{out}}$ or $\hat{F}''_{\mathsf{out}}$, the graphs match and thus the joined graph also matches as is shown in ⑥. However, because of the folding and the branch condition, the side constraints do not match and thus a join is computed in the abstract domain for sets. Because the domain is sufficiently precise, the set constraints shown in ⑥ are derived. Thus join is implemented by graph matching intertwined with queries and join operations in the abstract domain for sets.

At the end of the loop body, it is necessary to summarize the iteration element $\hat{f}$ into the already-visited set $\hat{F}_{\mathsf{out}}$. This allows the analysis to progress and it allows checking



```
for(var p in s)
{
  if(p in c) {
    r[p] = "conflict";
  } else {
    r[p] = s[p];
  }
}
```

1
i
2
3
4
5
6
r
s
c
p
notⁱ
undef
'conflict'
$\hat{F}_{in} = \hat{F}_s \wedge \hat{F}_{out} = \emptyset$
$\hat{F}_{in} \uplus \hat{F}_{out} = \hat{F}_s \wedge \hat{F}'_r = \hat{F}_r \setminus \hat{F}_{out} \wedge \hat{F}'_{out} \uplus \hat{F}''_{out} = \hat{F}_{out} \wedge \hat{F}''_{out} \cap \hat{F}_c = \emptyset \wedge \hat{F}'_{out} \subseteq \hat{F}_c$
$\hat{F}_{in} \uplus \hat{f} \uplus \hat{F}_{out} = \hat{F}_s \wedge \hat{F}'_r = \hat{F}_r \setminus \hat{F}_{out} \wedge \hat{F}'_{out} \uplus \hat{F}''_{out} = \hat{F}_{out} \wedge \hat{F}''_{out} \cap \hat{F}_c = \emptyset \wedge \hat{F}'_{out} \subseteq \hat{F}_c \wedge \hat{f} \subseteq \hat{F}_c$
$\hat{F}_{in} \uplus \hat{f} \uplus \hat{F}_{out} = \hat{F}_s \wedge \hat{F}'_r = \hat{F}_r \setminus (\hat{F}_{out} \uplus \hat{f}) \wedge \hat{F}'_{out} \uplus \hat{F}''_{out} = \hat{F}_{out} \wedge \hat{F}''_{out} \cap \hat{F}_c = \emptyset \wedge \hat{F}'_{out} \subseteq \hat{F}_c \wedge \hat{f} \subseteq \hat{F}_c$
$\hat{F}_{in} \uplus \hat{f} \uplus \hat{F}_{out} = \hat{F}_s \wedge \hat{F}'_r = \hat{F}_r \setminus \hat{F}_{out} \wedge \hat{F}'_{out} \uplus \hat{F}''_{out} = \hat{F}_{out} \wedge \hat{F}''_{out} \cap \hat{F}_c = \emptyset \wedge \hat{F}'_{out} \subseteq \hat{F}_c \wedge \hat{f} \cap \hat{F}_c = \emptyset$
$\hat{F}_{in} \uplus \hat{f} \uplus \hat{F}_{out} = \hat{F}_s \wedge \hat{F}'_r = \hat{F}_r \setminus (\hat{F}_{out} \uplus \hat{f}) \wedge \hat{F}'_{out} \uplus \hat{F}''_{out} = \hat{F}_{out} \wedge \hat{F}''_{out} \cap \hat{F}_c = \emptyset \wedge \hat{F}'_{out} \subseteq \hat{F}_c \wedge \hat{f} \cap \hat{F}_c = \emptyset$
$\hat{F}_{in} \uplus \hat{f} \uplus \hat{F}_{out} = \hat{F}_s \wedge \hat{F}'_r = \hat{F}_r \setminus (\hat{F}_{out} \uplus \hat{f}) \wedge \hat{F}'_{out} \uplus \hat{F}''_{out} = \hat{F}_{out} \uplus \hat{f} \wedge \hat{F}''_{out} \cap \hat{F}_c = \emptyset \wedge \hat{F}'_{out} \subseteq \hat{F}_c$

**Fig. 2** – Final iteration of analysis of the example loop from the Introduction. The loop invariant (i) shows the three inferred partitions of attr(r), and the set constraints (on the right) relate those three partitions to the partitions originally found in the three objects.

if the resulting state is contained in the loop invariant. The summarization process is a rewrite process where the partition $\hat{f}$ in $\hat{a}_3$ is merged with the partition $\hat{F}_{\mathsf{out}}$ and $\hat{F}_{\mathsf{out}} \uplus \hat{f}$ is rewritten with $\hat{F}_{\mathsf{out}}$ in the side constraints. The containment checking is similar to the join algorithm and proceeds by intertwined graph matching and set domain containment queries. In this case, the result of summarization matches the loop invariant ⓘ and thus the iteration process is complete and the loop invariant is inductive.

To find the loop invariant, HOO constructed new partitions (Section 3) through attribute materialization and updated them with strong updates (Section 4). Then it related those partitions with the iteration-progress variable $\hat{F}_{\mathsf{out}}$ by summarization (Section 5). As a result, HOO determined it did not need more partitions to express the loop invariant and that the result object `r` was related to the source object `s` through three partitions of attr(`r`).

## 3 Abstraction of Dynamic Language Heaps

In this section, we define the HOO abstraction. The HOO abstraction abstracts concrete dynamic language program states. A concrete program state $\sigma$ has the following definition:

$$\sigma : \mathrm{C} = \mathrm{Addr} \overset{\text{fin}}{\rightarrow} \mathrm{OMap} \times \mathrm{Value}_{\perp} \qquad o : \mathrm{OMap} = \mathrm{Attr} \overset{\text{fin}}{\rightarrow} \mathrm{Value}$$

Concrete states are finite maps from heap addresses (Addr) to concrete objects. A concrete object consists of two parts. The first part is the object mapping (OMap) that is a finite map from attributes (Attr) to values (Value). The second part is an optional value that is returned when an undefined attribute is read.

The HOO abstraction represents sets of concrete states with a finite disjunction of abstract states, that each consist of a heap graph and set constraints represented using an abstract domain for sets. Formally the HOO abstraction is the following:

**Definition 1 (Abstract State).** *An* abstract state $\Sigma \in \widehat{C}$ *is a pair of an abstract heap graph* $\hat{H}$ *and an element of an abstract domain for sets* $\hat{S}$. *The syntax of abstract heap graphs is*

$$\hat{H} ::= \textsc{Emp} \mid \textsc{True} \mid \hat{H} * \hat{H} \mid \hat{A} \cdot \hat{F} \mapsto \hat{V} \mid \hat{A} \cdot \mathsf{noti} \mapsto \hat{V}$$

*where symbols* $\hat{A}$, $\hat{F}$, *and* $\hat{V}$ *represent sets of addresses, attributes, and values respectively. We also use symbols* $\hat{a}$, $\hat{f}$, *and* $\hat{v}$ *to represent singleton sets of address, attributes, and values. The symbols for addresses and attributes are also symbols for values:*

$$\hat{A} \in \widehat{\mathrm{Addr}} \qquad \hat{F} \in \widehat{\mathrm{Attr}} \qquad \hat{V} \in \widehat{\mathrm{Value}} = \widehat{\mathrm{Addr}} \cup \widehat{\mathrm{Attr}} \cup \cdots$$

The resulting abstract domain is a reduced product [8] between a heap abstract domain element $\hat{H}$ and a set abstract domain element $\hat{S}$. The set domain is used to represent relationships between sets of attributes of objects. The information from the set domain affects points-to facts $\hat{A} \cdot \hat{F} \mapsto \hat{V}$ by constraining the sets of addresses $\hat{A}$, attributes $\hat{F}$, and values $\hat{V}$. Therefore the meaning of a HOO abstract state is closely tied to the meaning of set constraints. Since HOO is parametric with respect to the abstract domain for sets, its concretization is given in terms of a concretization for the set domain $\gamma(\hat{S})$:

$$\gamma(\hat{H}, \hat{S}) \stackrel{\text{def}}{=} \{ (\eta, \sigma) \mid (\eta, \sigma) \in \gamma(\hat{H}) \wedge \eta \in \gamma(\hat{S}) \}$$
$$\text{where} \quad \eta : \mathrm{E} = \widehat{\mathrm{Value}} \rightharpoonup \wp(\mathrm{Value})$$

The $\eta$ is a valuation function that maps value symbols (including address and attribute symbols) to sets of concrete values. The set domain restricts the $\eta$ function, which in turn restricts the concrete state $\sigma$ through the concretization of the heap. If $\_$ is a placeholder for unused existentially quantified variables, the concretization of the heap is defined as follows:

$$
\begin{array}{ll}
\gamma(\textsc{Emp}) & \overset{\text{def}}{=} \{\eta,\sigma \mid \mathrm{Dom}(\sigma)=\emptyset\} \\
\gamma(\textsc{True}) & \overset{\text{def}}{=} \mathrm{E}\times\mathrm{C} \\
\gamma(\hat{A}\cdot\hat{F}\mapsto\hat{V}) & \overset{\text{def}}{=} \left\{\eta,\sigma \,\middle|\, \forall a\in\eta(\hat{A}),f\in\eta(\hat{F}).\,\exists v\in\eta(\hat{V}),o.\,(o,\_)=\sigma(a)\wedge v=o(f)\right\} \\
\gamma(\hat{A}\cdot\mathsf{noti}\mapsto\hat{V}) & \overset{\text{def}}{=} \left\{\eta,\sigma \,\middle|\, \forall a\in\eta(\hat{A}).\,\exists v\in\eta(\hat{V}).\,(\_,v)=\sigma(a)\right\} \\
\gamma(\hat{H}_1*\hat{H}_2) & \overset{\text{def}}{=} \left\{\eta,\sigma \,\middle|\, \exists\sigma_1,\sigma_2.\,(\eta,\sigma_1)\in\gamma(\hat{H}_1)\wedge(\eta,\sigma_2)\in\gamma(\hat{H}_2)\wedge\sigma=\sigma_1\otimes\sigma_2\right\}
\end{array}
$$

The concretization of a points-to fact can represent part of many objects. The base addresses of the objects are retrieved from the valuation $\eta(\hat{A})$, but only the attributes retrieved from the valuation $\eta(\hat{F})$ are considered by this points-to fact. HOO uses an *attribute splitting* model similar to JStar [27] or Xisa [4], thus not every attribute of every object in $\eta(\hat{A})$ is represented in $\eta(\hat{F})$. Because each of the symbols $\hat{A}$ is a set, each abstract address may be a summary, but if the set domain can represent singletons [10, 28], these need not always be summaries.

The points-to fact for the default value $\hat{A}\cdot\mathsf{noti}\mapsto\hat{V}$ restricts the default value for each object in $\eta(\hat{A})$. These default value points-to facts serve a dual purpose, however. Because of the field splitting model, not all objects must have all of their attributes in a formula. The presence of a default points-to fact indicates that all of the objects of $\eta(\hat{A})$ are *complete*; they have all of their attributes represented in the formula. *Incomplete* objects may not have all of their attributes represented in the formula and thus abstract transfer functions may only access attributes that must be in the known parts of the object (see Section 4).

The separating conjunction has mostly the standard semantics [29]. Because objects can be split and the attributes are not fixed, we must define the composition $\otimes$ of two separate concrete states differently:

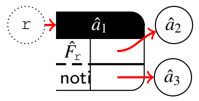
$$
\sigma_1\otimes\sigma_2=\lambda a.\begin{cases}
\sigma_1(a) & a\in\mathrm{Dom}(\sigma_1)\setminus\mathrm{Dom}(\sigma_2) \\
\sigma_2(a) & a\in\mathrm{Dom}(\sigma_2)\setminus\mathrm{Dom}(\sigma_1) \\
\begin{pmatrix} o_1\oplus o_2, \\ d_1\boxplus d_2 \end{pmatrix} & \begin{array}{l}(o_1,d_1)=\sigma_1(a)\\(o_2,d_2)=\sigma_2(a)\\a\in\mathrm{Dom}(\sigma_1)\cap\mathrm{Dom}(\sigma_2)\end{array}
\end{cases}
$$

$$
o_1\oplus o_2=\lambda s.\begin{cases}
o_1(f) & f\in\mathrm{Dom}(o_1)\setminus\mathrm{Dom}(o_2) \\
o_2(f) & f\in\mathrm{Dom}(o_2)\setminus\mathrm{Dom}(o_1)
\end{cases}
$$

Separate objects are composed trivially, but objects that have been split have their object maps composed using object map composition $\oplus$. This is only defined if there are disjoint attributes in each partial object map. Additionally, default values are composed with $\boxplus$ which yields the non-bottom value if possible and is undefined for two non-bottom values.

*Graphical Notation* In most of this paper, we use a graphical notation to help ease understanding. This notation can be translated to the formalization given in this section. In the graphical notation, a single circle represents an object address. If that circle is labeled with a $\hat{a}_n$, $\hat{f}_n$, or $\hat{v}_n$ the object is a singleton address, attribute or value respectively and thus corresponds to a single concrete value. If that circle is labeled with a $\hat{A}_n$, $\hat{F}_n$, or $\hat{V}_n$ and has a double border, the object is a summary. If that circle is labeled with a program variable, it represents a singleton stack location. Objects with fields are represented using the table notation, where each row corresponds to a points-to fact starting from a base address from the set $\hat{A}_n$.

*Example 1 (Graphical Notation Equivalence).* The following graphical notation and logical notation are equivalent. We use the unit attribute $()$ to represent the points-to relationship from the stack variable `r` to the singleton object $\hat{a}_1$.



equivalent to

$$\begin{array}{l} \mathtt{r}\cdot\{()\}\mapsto\hat{a}_1 \\ *\hat{a}_1\cdot\hat{F}_{\mathtt{r}}\mapsto\hat{a}_2 \\ *\hat{a}_1\cdot\mathsf{noti}\mapsto\hat{a}_3 \end{array}$$

## 4 Materialization and Transfer Functions

To precisely analyze programs that manipulate values in summaries, it is necessary to materialize individual elements from the summaries. Materialization occurs in execution of transfer functions in the language of commands $c$ that represents the core behaviors for open-object manipulation in dynamic languages:

$$\begin{array}{rll} c ::= & \mathbf{let}\,x=\mathrm{attr}(x_1) \mid \mathbf{let}\,x=\mathrm{choose}(x_1) & \rbrace\text{set operations} \\ & \mid \mathbf{let}\,x=x_1\cup x_2 \mid \mathbf{let}\,x=x_1\setminus x_2 & \\ & \mid \mathbf{let}\,x=x_1[x_2] \mid x_1[x_2]:=x_3 \mid \mathbf{for}\,x_1\,\mathbf{in}\,x_2\,\mathbf{do}\,c & \rbrace\text{object operations} \\ & \mid \mathbf{let}\,x=\mathbf{new}\{\} \mid c_1;c_2 \mid \mathbf{while}\,e\,\mathbf{do}\,c \mid \mathbf{let}\,x=e & \rbrace\text{standard operations} \end{array}$$

This section is concerned with load and store object operations because these operations require attribute materialization, which is mandatory for inferring precise relationships between objects with unknown attributes. Aside from **`for-in`**, which is defined in the next section, other operations, including choose$(x_1)$, which selects a singleton set from a set and attr$(x_1)$, which gets the union of all attributes of an object, are straightforward and given in the technical report [11].

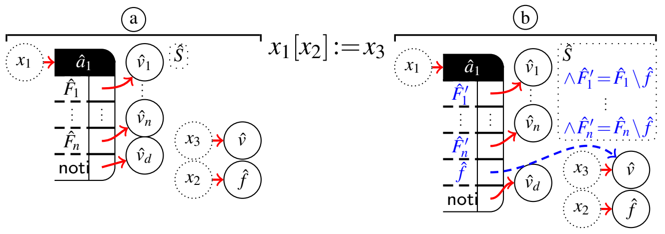
The concrete semantics of load **let** $x=x_1[x_2]$ and store $x_1[x_2]:=x_3$ are straightforward. They look up the object $x_1$, then try to find the given attribute $x_2$. Load binds to $x$ the value that corresponds to the attribute if it is found, otherwise it binds the default value for the object. Store removes the given attribute if it is found and adds a new attribute that corresponds to the right-hand side $x_3$.

To perform loads and stores on abstract objects the abstract transformers for load and store must determine how to manipulate and utilize the partitions on the accessed object. The process of transforming an object so that it has precisely the partitions necessary for performing a particular load or store is attribute materialization.

Concrete and abstract transfer functions are defined over the command language $c$. Concrete transformers $[\![c]\!]:\mathrm{C}\to\mathrm{C}$ transition a single concrete state to a single concrete state. Abstract transformers $\widehat{[\![c]\!]}:\widehat{\mathrm{C}}\to\wp(\widehat{\mathrm{C}})$ (shown as Hoare triples [20] with the graphical notation), however, transition a single abstract state to a set of abstract states representing a disjunction. This disjunction capability is used in transfer functions that perform case splits, such as the load transfer function. In the implementation of HOO, we use a disjunctive domain combinator to manage these sets.

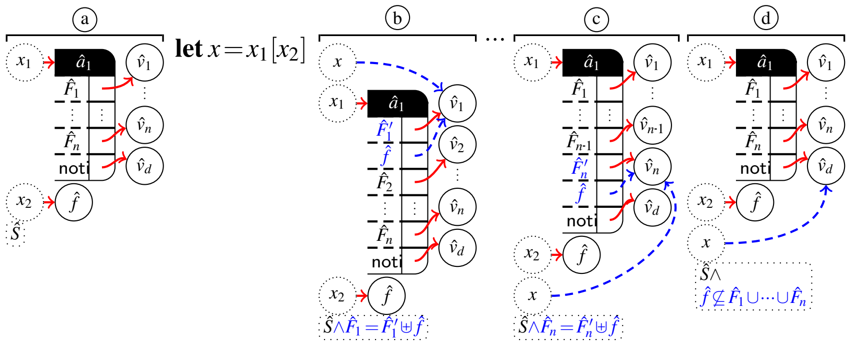
It is possible to implement transfer functions that manipulate complete, incomplete, summary, and singleton objects. Here we define the store and load transfer functions for complete singleton objects. For incomplete objects, there are separate transfer functions: before a materialization can occur, it must be proven that the attribute already exists in the object. This ensures that attributes that are defined in the missing part of the object cannot be read or overwritten by any operations. When operating on a summary object, a singleton must first be materialized. This materialization is trivially defined through case splits that result in finite disjunctions.

***Attribute Materialization for Store:*** Attribute materialization for store operations is simple. Since the value of the particular attribute is about to be overwritten, there is no need to preserve the original value. The implementation of store is the following:


a
$x_1[x_2] := x_3$
b
$\hat{S}$
$\wedge \hat{F}'_1 = \hat{F}_1 \setminus \hat{f}$
$\wedge \hat{F}'_n = \hat{F}_n \setminus \hat{f}$
noti

Store looks up the corresponding objects to $x_1$, $x_2$, and $x_3$ in ⓐ, which in this case are $\hat{a}_1$, $\hat{f}$, and $\hat{v}$ respectively. Attribute materialization then iterates through each partition in $\hat{a}_1$ and reconstructs the partition by removing $\hat{f}$ from the partition. If $\hat{f}$ was not already present in the partition, this represents no change, otherwise it explicitly removes $\hat{f}$. Finally, after all of the existing partitions have been reconstructed, a new partition for $\hat{f}$ is created and it is pointed to the stored value $\hat{v}$ giving ⓑ. By performing this attribute materialization, we have guaranteed that subsequent reads of the same property $\hat{f}$, even if we do not know its concrete value, will be directed to $\hat{f}$, and thus store performs strong updates.

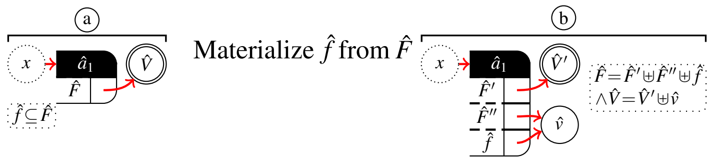
***Attribute Materialization for Load:*** Attribute materialization for load is similar to store. The key difference is that there is a possible result for each partition of the read object. The HOO abstract domain uses a finite disjunction to represent the result of this case split:


a
**let** $x = x_1[x_2]$
b
c
d
noti
$\hat{S} \wedge \hat{F}_1 = \hat{F}'_1 \uplus \hat{f}$
$\hat{S} \wedge \hat{F}_n = \hat{F}'_n \uplus \hat{f}$
$\hat{S} \wedge \hat{f} \not\subseteq \hat{F}_1 \cup \cdots \cup \hat{F}_n$

A load operation must determine which, if any, of the partitions the attribute $\hat{f}$ is in. In the worst case, it could be in any of the partitions and therefore a result must be considered for each case. In each non-noti case, $\hat{f}$ is constrained to be in that particular partition and therefore in no other partition. If this is inconsistent under the current analysis state, the abstract state will become bottom for that case and it can be dropped. The noti partition, which implicitly represents all attributes not currently in the object, must be considered as a possible source for materialization if there is a chance the attribute does not already exist in the object. Such a materialization does not explicitly cause any repartitioning because noti still represents all of the not present attributes (which now does not include $\hat{f}$).

If the values that are being loaded (in this case $\hat{v}_1,\cdots,\hat{v}_n,\hat{v}_d$) are not singleton values, the load operation must also materialize one value from that summary. When materializing from a summary object, additional partitions can be generated. For each object that has a partition that maps to the summary, that partition must be split into two parts: one that maps to a new summary and one that maps to the singleton that was materialized. While it is possible that these case splits introduced by load could become prohibitive, we have not found this to be a significant problem. Typically unknown attributes are not completely unknown and thus limit case splits or the number of partitions for an object is sufficiently small that these case splits do cause significant problems. If the precision provided by the case splits is unneeded, the resulting states can be joined to eliminate cases.

*Example 2 (Store with summary values).* When loading from an attribute $\hat{f}$ that is contained in a partition $\hat{F}$ of an object $\hat{a}$ that maps to a summary $\hat{V}$, additional partitions are produced. The result contains three partitions instead of two. Some attributes from $\hat{F}$ map to $\hat{V}'$ and some map to $\hat{v}$. Therefore, while the analysis knows that $\hat{f}$ maps to $\hat{v}$ because that is why it chose to materialize $\hat{v}$, it does not know that other attributes of $\hat{F}$ do not also map to $\hat{v}$. Therefore, it splits the remainder of $\hat{F}$ into two partitions: one $\hat{F}'$ that maps to the remainder of the values $\hat{V}'$ and another $\hat{F}''$ that maps to the materialized value $\hat{v}$.


a
x
$\hat{a}_1$
$\hat{F}$
$\hat{V}$
$\hat{f}\subseteq\hat{F}$
Materialize $\hat{f}$ from $\hat{F}$
b
x
$\hat{a}_1$
$\hat{F}'$
$\hat{F}''$
$\hat{f}$
$\hat{V}'$
$\hat{v}$
$\hat{F}=\hat{F}'\uplus\hat{F}''\uplus\hat{f}$
$\wedge\hat{V}=\hat{V}'\uplus\hat{v}$

**Theorem 1 (Soundness of transfer functions).** *Transfer functions are sound because for any command c, the following property holds:*

$$\forall(\hat{H},\hat{S})\in\widehat{C},\sigma\in\gamma(\hat{H},\hat{S}),\bar{\Sigma}\subseteq\widehat{C}.$$
$$\bar{\Sigma}=\widehat{[\![c]\!]}(\hat{H},\hat{S})\Rightarrow\exists(\hat{H}',\hat{S}')\in\bar{\Sigma}.\,[\![c]\!]\sigma\in\gamma(\hat{H}',\hat{S}')$$

## 5 Automatic Invariant Inference

In this section we give the join, widening, and inclusion check algorithms that are required for automatically and soundly generating program invariants. Here the focus is inferring loop invariants for **`for-in`** loops — the primary kind of loop for object-manipulation. The analysis of **`for-in`** loops first translates these loops into **`while`** loops. This allows HOO to follow the standard abstract interpretation procedure for loops, while introducing iteration-progress variables to aid the analysis in inferring precise loop invariants.

```
for $x_1$ in $x_2$ do $c \stackrel{\text{def}}{=}$
    let $s = $ new$\{\}$;
    $s$['in'] := attr($x_2$);
    $s$['out'] := $\emptyset$;
    while $s$['in'] $\neq \emptyset$ do
        let $x_1$ = choose($s$['in']);
        $s$['in'] := $s$['in'] $\setminus \{x_1\}$
        $c$;
        $s$['out'] = $s$['out'] $\cup \{x_1\}$
```

These iteration-progress variables are introduced in the translation process shown in the inset figure. For the object being iterated over $x_2$, the $s$['in'] variable keeps track of attributes that have not yet been visited by the loop, while $s$['out'] keeps track of attributes that have already been visited by the loop. To keep these variables up to date, the translation employs the set manipulating commands introduced in Section 4.

Once translated, HOO takes advantage of $s$['in'] and $s$['out'] to represent relations between partitions of attributes. Adding these ghost variables, allows partitions to be equal to a function of the already visited portion attr$(x_2)$. On the exit of the loop, $s$['in'] is the empty set and $s$['out'] is attr$(x_2)$, so partitions related to $s$['out'] are now related to attr$(x_2)$.

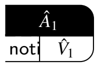
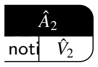
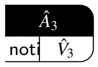
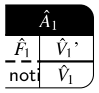
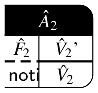
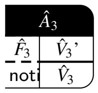
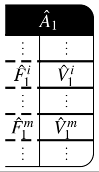
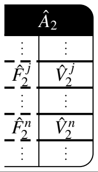
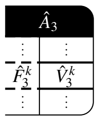
These iteration-progress variables are essential for performing strong updates. When analyzing an iteration of a loop, partitions that arise from attribute materialization arise simultaneously with partitions that arise in iteration-progress variables. Thus these partitions become related and even when partitions from attribute materialization must be summarized, the relationship with the iteration progress variable is maintained. The summarization process occurs as part of join and widening.

***Join Algorithm:*** The join algorithm takes two abstract states $\hat{H}_1,\hat{S}_1$ and $\hat{H}_2,\hat{S}_2$ and computes an overapproximation of all program states described by each of these abstract states. When joining abstractions of memory, the algorithm must match objects in $\hat{H}_1$ and objects in $\hat{H}_2$ to objects in a resulting abstract memory $\hat{H}_3$. This matching of objects can be described by two mapping functions $M_1$ and $M_2$, where $M_1 : \widehat{\text{Addr}}_1 \xrightarrow{\text{fin}} \widehat{\text{Addr}}_3$ maps symbols from $\hat{H}_1$ to symbols from $\hat{H}_3$ and $M_2 : \widehat{\text{Addr}}_2 \xrightarrow{\text{fin}} \widehat{\text{Addr}}_3$ maps symbols from $\hat{H}_2$ to symbols from $\hat{H}_3$. However, because HOO abstracts open objects, the join algorithm must match partitions of objects as well. This matching is represented with a relation $P_J \subseteq \wp(\widehat{\text{Attr}}_1) \times \wp(\widehat{\text{Attr}}_2) \times \widehat{\text{Attr}}_3$ that relates sets of partitions from objects in $\hat{H}_1$ and $\hat{H}_2$ to partitions in $\hat{H}_3$. Because partitions can be split and because new, empty partitions can be created, join can produce an unbounded number of partitions.

The fundamental challenge for the HOO abstraction's join algorithm is computing these symbol matchings $M_1$, $M_2$, and $P_J$. To construct the matchings, the join algorithm begins at the symbolic addresses of stack allocated variables. It adds equivalent variables from the three graphs to $M_1$ and $M_2$, then it begins an iterative process. Starting from a matching that already exists in $M_1$ and $M_2$, it derives additional matchings that are potential consequences. To derive these additional matchings, a template system is used. The templates are shown in Table 1. These templates consume corresponding parts of a memory abstraction, producing a resultant memory abstraction that holds under the matchings. This iterative process is applied until no more templates can be applied. Any remaining heap at this point results in TRUE being added to the result. The result of join is complete matchings $M_1$, $M_2$, and $P_J$, as well as, a memory abstraction $\hat{H}_3$. To get the resulting set abstraction $\hat{S}_3$, the sets are joined under the same matchings, where multiple matchings are interpreted as a union.

There are three templates described in Table 1. The first trivially joins any two empty objects into an empty object. The default values are subsequently matched. The second

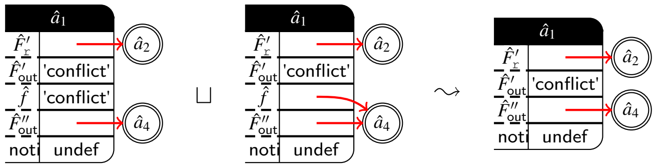
**Table 1** – Join templates match objects in two abstract heaps, producing a third heap that overapproximates both. Matchings $M_1$, $M_2$, $P_J$ are generated on the fly and used in joining the set domain after the heaps are joined.

| Prerequisites | $\hat{H}_1,\hat{S}_1$ | $\sqcup$ | $\hat{H}_2,\hat{S}_2$ | $\rightsquigarrow$ | Result | |
|---|---|---|---|---|---|---|
| $M_1(\hat{A}_1)=\hat{A}_3$<br>$M_2(\hat{A}_2)=\hat{A}_3$ |  | $\sqcup$ |  | $\rightsquigarrow$ |  | $M_1(\hat{V}_1)=\hat{V}_3$<br>$M_2(\hat{V}_2)=\hat{V}_3$ |
| $M_1(\hat{A}_1)=\hat{A}_3$<br>$M_2(\hat{A}_2)=\hat{A}_3$ |  $\hat{A}_1$<br>$\hat{F}_1$<br>$\hat{V}_1$'<br>noti<br>$\hat{V}_1$ | $\sqcup$ |  $\hat{A}_2$<br>$\hat{F}_2$<br>$\hat{V}_2$'<br>noti<br>$\hat{V}_2$ | $\rightsquigarrow$ |  $\hat{A}_3$<br>$\hat{F}_3$<br>$\hat{V}_3$'<br>noti<br>$\hat{V}_3$ | $M_1(\hat{V}_1)=\hat{V}_3,\ M_2(\hat{V}_2)=\hat{V}_3$<br>$M_1(\hat{V}_1')=\hat{V}_3',\ M_2(\hat{V}_2')=\hat{V}_3'$<br>$(\{\hat{F}_1\},\{\hat{F}_2\},\hat{F}_3)\in P_J$ |
| $M_1(\hat{A}_1)=\hat{A}_3$<br>$M_2(\hat{A}_2)=\hat{A}_3$<br>remainder of object matches |  $\hat{A}_1$<br>$\hat{F}_1^i$<br>$\hat{V}_1^i$<br>$\hat{F}_1^m$<br>$\hat{V}_1^m$ | $\sqcup$ |  $\hat{A}_2$<br>$\hat{F}_2^j$<br>$\hat{V}_2^j$<br>$\hat{F}_2^n$<br>$\hat{V}_2^n$ | $\rightsquigarrow$ |  | $(\{\hat{F}_1^i,\cdots,\hat{F}_1^m\},\{\hat{F}_2^j,\cdots\hat{F}_2^n\},\hat{F}_3^k)\in P_J$<br>$M_1(\hat{V}_1^i)=\hat{V}_3^k,\ M_2(\hat{V}_2^j)=\hat{V}_3^k$<br>$\vdots \qquad \vdots$<br>$M_1(\hat{V}_1^m)=\hat{V}_3^k,\ M_2(\hat{V}_2^n)=\hat{V}_3^k$ |

template joins any two objects that have only one partition. The values from that partition are added to the mapping as well as the default values. The last template is parametric. If some number of partitions can be matched with some number of partitions then those can all be merged into a single partition in the result. This template requires applying other rules to complete the joining of the objects. If it is unknown how to match partitions for all of an object, this template allows matching part of the object. If the result is that remaining partitions are single partitions, even if there is no natural way to match them, they will be matched by applying template two.

*Example 3 (Joining objects).* Here we join $\hat{a}_1$ objects from the overview example at program points ③ and ⑤ to get the result shown at ⑥. To compute the join we construct matchings $M_1$, $M_2$, and $P_J$. Initially $M_1=[\hat{a}_1 \mapsto \hat{a}_1]$, $M_2=[\hat{a}_1 \mapsto \hat{a}_1]$, and $P_J=\emptyset$. If we were to match $\hat{F}'_{\text{out}}$ with $\hat{F}'_{\text{out}}$ or $\hat{F}''_{\text{out}}$ with $\hat{F}''_{\text{out}}$, we would get an imprecise join because we would be forced to match $\hat{f}$ with itself even though it has two values that should not be joined. Instead, we apply the third template to merge partitions with like values, thus merging $\hat{f}$ with $\hat{F}'_{\text{out}}$ in ③ and with $\hat{F}''_{\text{out}}$ in ⑤. Since the only remaining partition is $\hat{F}'_{\mathtt{r}}$, we match $\hat{F}'_{\mathtt{r}}$ and $\hat{F}'_{\mathtt{r}}$ giving the following matchings and join result:

$$M_1=[\hat{a}_1 \mapsto \hat{a}_1, \hat{a}_2 \mapsto \hat{a}_2, \hat{a}_4 \mapsto \hat{a}_4]$$
$$M_2=[\hat{a}_1 \mapsto \hat{a}_1, \hat{a}_2 \mapsto \hat{a}_2, \hat{a}_4 \mapsto \hat{a}_4]$$
$$P_J=\{(\{\hat{F}'_{\mathtt{r}}\},\{\hat{F}'_{\mathtt{r}}\},\hat{F}'_{\mathtt{r}}),\ (\{\hat{F}'_{\text{out}},\hat{f}\},\{\hat{F}'_{\text{out}}\},\hat{F}'_{\text{out}}),\ (\{\hat{F}''_{\text{out}}\},\{\hat{F}''_{\text{out}},\hat{f}\},\hat{F}''_{\text{out}})\}$$



***Widening algorithm:*** In HOO, the join and widening algorithms are nearly identical. However, unlike join, widening must select matchings that ensure convergence of the analysis, by guaranteeing that the number of partitions does not grow unboundedly and that the arrangement of the partitions is fixed (i.e. there is no oscillation in which partitions are matched during widening). While there are many possible approaches that meet these criteria, we utilize allocation site information to resolve decisions during the matching process. Only objects from the same allocation site may be matched, which causes only attribute sets whose corresponding values are from the same allocation site to be matched. To ensure convergence, after some number of iterations, all objects from the same allocation site can be forced to be matched. This bounds the partitions per abstract object to one per allocation site and bounds the number of abstract objects to one per allocation site, so as long as the underlying set domain converges on an abstraction for each partition, the analysis will converge.

***Inclusion Check Algorithm:*** Inclusion checking determines if an abstract state is already described by another abstract state. The process for deciding if an inclusion holds is similar to the join processes. If $M, P_I \vdash \hat{H}_a, \hat{S}_a \sqsubseteq \hat{H}_b, \hat{S}_b$, all concrete states described by $\hat{H}_a, \hat{S}_a$ must be contained in the set of all concrete states described by $\hat{H}_b, \hat{S}_b$. It works in a fashion similar to join by constructing matchings $M$ and $P_I$ from symbols in $\hat{H}_a, \hat{S}_a$ to symbols in $\hat{H}_b, \hat{S}_b$. It employs the same methodology as join. Objects are matched, one-by-one, until no more matches can be made. This matching builds up the mapping $M$ that is then used for an inclusion check in the set domain. If the mapping was successfully constructed and the inclusion check holds in the set domain, the inclusion check holds on the HOO domain. The templates for augmenting the mapping are essentially the same as those for join shown in Table 1, except with only $M_1$ and with $P_I$ only using the first and third components and where $\hat{H}_2, \hat{S}_2$ is ignored with $\hat{H}_1, \hat{S}_1$ corresponding to $\hat{H}_a, \hat{S}_a$ and the result corresponding to $\hat{H}_b, \hat{S}_b$.

**Theorem 2 (Join Soundness).** *Join is sound under matchings $M_1$, $M_2$, $P_J$ because*

$$\text{If } M_1, M_2, P_J \vdash \hat{H}_1, \hat{S}_1 \sqcup \hat{H}_2, \hat{S}_2 \rightsquigarrow \hat{H}_3, \hat{S}_3 \text{ then}$$
$$\forall \sigma, \eta.\, (\eta, \sigma) \in \gamma(\hat{H}_1, \hat{S}_1) \lor (\eta, \sigma) \in \gamma(\hat{H}_2, \hat{S}_2) \Rightarrow \exists \eta_3.\, (\eta_3, \sigma) \in \gamma(\hat{H}_3, \hat{S}_3)$$

We do not state properties other than soundness due to the dependence of HOO's behavior on its instantiation. Because of the non-trivial interaction between the set domain and HOO, properties of HOO are affected by properties of the set domain. More precise set domain operations typically yield more precision in HOO. Additionally, the choice of heuristics for template application can affect the results of join, widening, and inclusion check, thus leading to a complex dependency between precision and heuristics. While this dependence can affect many properties, it does not affect soundness.

## 6 Precision Evaluation

In this section we test several hypotheses: first, that HOO is fast enough to be useful; second, that HOO is at least as precise as other open-object abstractions when objects have unknown attributes; and third, that HOO infers partitions and relations between partitions of unknown attributes precisely enough to verify properties of intricate object-manipulating programs. To investigate these hypotheses, we created a prototype implementation in OCaml and ran it on a number of small diagnostic benchmarks, each of which consists

**Table 2** – Analysis results of diagnostic benchmarks. Time compare analysis time excluding JVM startup time. Memory properties compares TAJS and HOO in verifying pointer properties. Object properties compares TAJS and HOO in verifying object structure properties. The # Props columns are the total number of properties of that kind.

| | Time (s) | | Memory Properties | | | Object Properties | | |
|---|---|---|---|---|---|---|---|---|
| Program | TAJS | HOO | TAJS | HOO | # Props | TAJS | HOO | # Props |
| `static` | 0.06 | 0.09 | 1 | 1 | 1 | 3 | 3 | 3 |
| `copy` | 0.13 | 0.02 | 1 | 1 | 1 | 0 | 3 | 3 |
| `filter` | 0.40 | 0.10 | 0 | 0 | 0 | 0 | 6 | 6 |
| `compose` | 0.71 | 0.54 | 0 | 0 | 0 | 0 | 7 | 7 |
| `merge` | 0.19 | 0.06 | 2 | 2 | 2 | 0 | 5 | 5 |

of one or more loops that manipulate open objects. These benchmarks are drawn from real JavaScript frameworks such as JQuery, Prototype.js, and Traits.js[3]. We chose them to test commonly occurring idioms that manipulate open objects in dynamic languages. To have properties to verify, we developed partial correctness specifications for each of the benchmarks. We then split the post-conditions of the specifications into a number of properties to verify that belong in one of two categories: memory properties assert facts about pointers (e.g., $\texttt{r} \neq \texttt{s}$), and object properties assert facts about the structure of objects (e.g., if the object at $\hat{a}_1$ has attribute $\hat{f}$, then object at $\hat{a}_2$ also has attribute $\hat{f}$).

We use these benchmarks to compare HOO with TAJS [21], which is currently the most precise (for open objects) JavaScript analyzer. Because TAJS is a whole-program analysis, it is not intended to verify partial correctness specifications and consequently, it adapts a traditional field-sensitive object representation for open objects. However, it employs several features to improve precision when unknown attribute are encountered during analysis: it implements a recency abstraction [1] to allow strong updates on straight-line code, and it implements correlation tracking [31] to allow statically known attributes to be iteratively copied using **`for-in`** loops.

In the results in Table 2, we find that TAJS and HOO are able to prove the same memory properties. The diagnostic benchmarks are not designed to exercise intricate memory structures, so all properties are provable with an allocation site abstraction. Because both TAJS and HOO use allocation site information, both prove all memory properties.

For object properties, HOO is always at least as precise as TAJS, and significantly more so when unknown attributes are involved. The `static` benchmark is designed to simulate the "best-case scenario" for whole program analyses: it supplies all attributes to objects before iterating over them. Here, TAJS relies on correlation tracking to prove all properties. HOO can also prove all of these properties. It infers a separate partition for each statically known attribute, effectively making it equivalent to TAJS's object abstraction.

Our other benchmarks iterate over objects where the attributes are unknown. Here, HOO proves all properties, while TAJS fails to prove any. TAJS's imprecision is unsurprising because correlation tracking does not work with unknown attributes and recency

[3] http://jquery.com, http://prototypejs.org, and http://traitsjs.org

abstraction does not enable strong updates in loops. HOO, on the other hand, succeeds because it infers partitions of object attributes and relates those partitions to other partitions. In the `copy` benchmark, attributes and values are copied one attribute at a time to a new object. HOO infers that after the iteration is complete, the attributes of both objects are equal. HOO can also verify the `filter` benchmark, which is the example presented throughout this paper that requires conditional and partial overwriting of objects. Additionally, HOO continues to be precise even when complex compositions are involved, as in the `compose` and `merge` benchmarks, which perform parallel and serial composition of objects. For these benchmarks HOO infers relationships between multiple objects and sequentially updates objects through multiple **`for-in`** loops.

On these benchmarks, HOO is often faster than TAJS, but this is likely due to TAJS's full support for JavaScript and the DOM and thus performance is really incomparable. Actually, HOO's performance is highly dependent on the efficiency of the underlying set domain due to the large number of set domain operations that HOO uses. However, despite not having a heavily optimized set domain, HOO analyzes these benchmarks quickly.

This evaluation demonstrates that HOO is effective at representing and verifying properties of open objects, both with statically known attributes and with entirely unknown attributes. Additionally it shows that HOO provides significant precision improvement over existing open-object abstractions when attributes are unknown and that HOO does not take a significant amount of time to verify complex properties.

## 7 Related Work

***Analyses for dynamic languages:*** Because one of the main features of dynamic languages is open objects, all analyses for dynamic languages must handle open objects to a degree. As opposed to directly abstracting open objects, TAJS [21, 22], WALA [31], and JSAI [19, 24] extend standard field-sensitive analyses to JavaScript by adding a summary field for all unknown attributes. They employ clever interprocedural analysis tricks to propagate statically known object attributes through loops and across function call boundaries. Consequently, with the whole program, they can often precisely verify properties of open-object manipulating programs. Without the whole program, these techniques lose precision because they conflate all unknown object attributes into a single summary field and weakly update it through loops.

***Analyses for containers:*** Because objects in dynamic languages behave similarly to containers, it is possible that a container analysis could be adapted to this task. Powerful container analyses such as [14] and [17] can represent and infer arbitrary partitions of containers. This is similar to HOO except that they do not use set abstractions to represent the partitions, but instead use SMT formulas and quantifier templates. For some applications these are excellent choices, but for dynamic languages where the key type of the containers is nearly always strings, this suffers. HOO can use abstract domains for sets [10, 28] and thus if these domains are parametric over their value types, HOO can support nearly any key-type abstraction.

Arrays and lists are restricted forms of containers on which there has been a significant amount of work [2, 9, 13, 16, 18, 23, 25]. The primary difference between arrays and more general containers and open objects is that arrays typically contain related values next to one another. Partitions of arrays are implicitly ordered and because array keys typically do not have gaps, partitions are defined using expressions that identify partition boundaries. Because open objects have gaps and are unordered, array analyses are not applicable. Regardless, array abstraction inspires the partitioning of open objects that we use.

***Decision procedures:*** In addition there are analyses that do not handle loops without annotations for both dynamic languages and containers. DJS [5, 6] is a flow-sensitive dependent type system for JavaScript. It can infer intermediate states in straight-line code, but it requires annotations for loops and functions. Similarly JuS [15] supports straight-line code for JavaScript. Jahob and its brethren [26] use a battery of different decision procedures to analyze containers and the heap together for Java programs. Finally, array decision procedures [3, 12] can be adapted to containers, but all of these approaches require significant annotation of non-trivial loop invariants to be effective on open-object-manipulating programs.

## 8 Conclusion and Future Work

In an effort to verify properties of incomplete, open-object-manipulating programs, we created the HOO abstract domain. It is capable of verifying complex object manipulations even when object attributes are completely unknown. While it is effective today, we want to extend it to allow inferring relationships between attributes and their corresponding values. Such relationships enable determining precisely which value in a summary is being materialized and proving properties about specific values, even when they are included in a summary. We plan to pursue such an extension as we believe that it could enable verification of programs that use open objects not only as objects, but also as containers.

**Acknowledgements.** Thank you to 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 the National Science Foundation under Grant Numbers CCF-1055066 and CCF-1218208. The research leading to these results has also received funding from the European Research Council under the FP7 grant agreement 278673, Project MemCAD.

## Bibliography

1. G. Balakrishnan and T. W. Reps. Recency-abstraction for heap-allocated storage. In *SAS*, pages 221–239, 2006.
2. A. Bouajjani, C. Dragoi, C. Enea, and M. Sighireanu. Abstract domains for automated reasoning about list-manipulating programs with infinite data. In *VMCAI*, pages 1–22, 2012.
3. A. R. Bradley, Z. Manna, and H. B. Sipma. What's decidable about arrays? In *VMCAI*, pages 427–442, 2006.
4. B.-Y. E. Chang and X. Rival. Relational inductive shape analysis. In *POPL*, pages 247–260, 2008.
5. R. Chugh, D. Herman, and R. Jhala. Dependent types for JavaScript. In *OOPSLA*, pages 587–606, 2012.
6. R. Chugh, P. M. Rondon, and R. Jhala. Nested refinements: a logic for duck typing. In *POPL*, pages 231–244, 2012.

7. P. Cousot and R. 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. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL*, pages 269–282, 1979.
9. P. Cousot, R. Cousot, and F. Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *POPL*, pages 105–118, 2011.
10. A. Cox, B.-Y. E. Chang, and S. Sankaranarayanan. QUIC graphs: Relational invariant generation for containers. In *ECOOP*, pages 401–425, 2013.
11. A. Cox, B.-Y. E. Chang, and X. Rival. Automatic analysis of open objects in dynamic language programs (extended). Technical report, University of Colorado Boulder, 2014.
12. L. M. de Moura and N. Bjørner. Generalized, efficient array decision procedures. In *FMCAD*, pages 45–52, 2009.
13. I. Dillig, T. Dillig, and A. Aiken. Fluid updates: Beyond strong vs. weak updates. In *ESOP*, pages 246–266, 2010.
14. I. Dillig, T. Dillig, and A. Aiken. Precise reasoning for programs using containers. In *POPL*, pages 187–200, 2011.
15. P. Gardner, S. Maffeis, and G. D. Smith. Towards a program logic for JavaScript. In *POPL*, pages 31–44, 2012.
16. D. Gopan, T. W. Reps, and S. Sagiv. A framework for numeric analysis of array operations. In *POPL*, pages 338–350, 2005.
17. S. Gulwani, B. McCloskey, and A. Tiwari. Lifting abstract interpreters to quantified logical domains. In *POPL*, pages 235–246, 2008.
18. N. Halbwachs and M. Péron. Discovering properties about arrays in simple programs. In *PLDI*, pages 339–348, 2008.
19. B. Hardekopf, B. Wiedermann, B. R. Churchill, and V. Kashyap. Widening for control-flow. In *VMCAI*, pages 472–491, 2014.
20. C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10): 576–580, 1969.
21. S. H. Jensen, A. Møller, and P. Thiemann. Type analysis for JavaScript. In *SAS*, pages 238–255, 2009.
22. S. H. Jensen, A. Møller, and P. Thiemann. Interprocedural analysis with lazy propagation. In *SAS*, pages 320–339, 2010.
23. R. Jhala and K. L. McMillan. Array abstractions from proofs. In *CAV*, pages 193–206, 2007.
24. V. Kashyap, J. Sarracino, J. Wagner, B. Wiedermann, and B. Hardekopf. Type refinement for static analysis of JavaScript. In *DLS*, pages 17–26, 2013.
25. L. Kovács and A. Voronkov. Finding loop invariants for programs over arrays using a theorem prover. In *FASE*, pages 470–485, 2009.
26. V. Kuncak. *Modular Data Structure Verification*. PhD thesis, EECS Department, Massachusetts Institute of Technology, February 2007.
27. M. J. Parkinson. *Local reasoning for Java*. PhD thesis, University of Cambridge, 2005.
28. T.-H. Pham, M.-T. Trinh, A.-H. Truong, and W.-N. Chin. Fixbag: A fixpoint calculator for quantified bag constraints. In *CAV*, pages 656–662, 2011.
29. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74, 2002.
30. G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of JavaScript programs. In *PLDI*, pages 1–12, 2010.
31. M. Sridharan, J. Dolby, S. Chandra, M. Schäfer, and F. Tip. Correlation tracking for points-to analysis of JavaScript. In *ECOOP*, pages 435–458, 2012.