Revisiting Recency Abstraction for JavaScript

Towards an Intuitive, Compositional, and Efficient Heap Abstraction

Jihyeok Park
KAIST
jhpark0223@kaist.ac.kr

Xavier Rival
CNRS, ENS, INRIA Paris, PSL*
Research University
Xavier.Rival@ens.fr

Sukyoung Ryu
KAIST
sryu.cs@kaist.ac.kr

Abstract

JavaScript is one of the most widely used programming languages. To understand the behaviors of JavaScript programs and to detect possible errors in them, researchers have developed several static analyzers based on the abstract interpretation framework. However, JavaScript provides various language features that are difficult to analyze statically and precisely such as dynamic addition and removal of object properties, first-class property names, and higher-order functions. To alleviate the problem, JavaScript static analyzers often use *recency abstraction*, which refines address abstraction by distinguishing recent objects from summaries of old objects. We observed that while recency abstraction enables more precise analysis results by allowing *strong updates* on recent objects, it is not monotone in the sense that it does not preserve the precision relationship between the underlying address abstraction techniques: for an address abstraction A and a more precise abstraction B, recency abstraction on B may not be more precise than recency abstraction on A.

Such an unintuitive semantics of recency abstraction makes its composition with various analysis sensitivity techniques also unintuitive. In this paper, we propose a new *singleton abstraction* technique, which provides a monotone refinement relationship of the underlying address abstraction. We prove the monotonicity of singleton abstraction and our prototype implementation shows promising results.

CCS Concepts • **Software and its engineering** → **General programming languages**; • **Theory of computation** → *Program analysis*

Keywords Address abstraction, recency abstraction, address partition

1. Introduction

JavaScript is one of the most widely used programming languages. It is now the 7th popular language [1] and it becomes the *de facto* language for web programming. In the ever-growing IoT era, the realm of JavaScript may expand even more [2] and understanding and detecting bugs in JavaScript programs are getting more important.

Recently, researchers have presented several static analyzers for JavaScript programs. SAFE [9], TAJS [6], and WALA [11] statically analyze JavaScript programs based on the abstract interpretation framework. Because they aim for sound static analysis, their analysis results are often imprecise. Thus, each analyzer develops its own analysis techniques to improve the analysis precision [3, 10–12].

For JavaScript static analysis, analyzing "object properties" precisely serves an important role in improving the analysis precision. First, because property names themselves are first-class values, imprecise analysis of property names lead to imprecise analysis of property accesses. Second, since object properties may be added or removed dynamically, precisely analyzing the existence of object properties is challenging. Imprecisely analyzing that a specific property may not exist in an object may result in reporting a false type error. Third, because JavaScript supports higher-order functions, the values of object properties may be functions, which implies that building control flow graphs precisely requires precise analysis of property accesses.

To analyze object properties more precisely, JavaScript static analyzers often use *recency abstraction* [4]. Note that one of main causes of the analysis imprecision is *weak update*, which updates the value of an object property to a join of its old value and a new value. To analyze such updates more precisely, recency abstraction distinguishes the most recently allocated objects from joined old objects and performs weak updates on joined old objects and *strong updates* that replace old values with new values on the recently allocated objects. Thus, recency abstraction enhances the analysis precision for the most recently allocated objects.

Recency abstraction is yet another address abstraction that divides a given (underlying) partition-based address abstraction into two parts: *old* and *recent* addresses. A partition-based address abstraction divides addresses into partitions and uses their powersets as its abstract domain. One example partition-based address abstraction is the *allocation-site abstraction*, which creates partitions by merging all objects created at the same allocation sites. For each partition, recency abstraction distinguishes a recent address that points to the most recently created objects and an old address that points to old objects, and it allows strong updates only on recent addresses. Consider the following code:

CONF'yy Month d–d, 20yy, City, ST, Country

© 20yy Copyright held by the owner/author(s). Publication rights licensed to ACM.
ISBN 978-1-nnnn-nnnn-n/yy/mm. . . $15.00

DOI: http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

ℓ_0 : $\mathbf{function}$ $\mathtt{f}()$ { $\mathbf{return}$ {}; };
ℓ_1 : $\mathbf{var}$ $\mathtt{x} = \mathtt{f}()$;
ℓ_2 : $\mathbf{var}$ $\mathtt{y} = \mathtt{f}()$;
ℓ_3 : $\mathtt{x.p} = 1$;
ℓ_4 : $\mathtt{y.p} = 2$;
ℓ_5 : $\mathtt{x.p} + \mathtt{y.p}$

Though it is contrived for the presentation brevity, it shows how recency abstraction for the allocation-site abstraction works succinctly. Since the values of x and y are objects created at the same allocation site, ℓ_0, both x and y have the same partition ℓ_0 in the allocation-site abstraction. Thus, at the end of the above code, both x.p and y.p have $\mathbf{undefined}$ (the default value for an absent property), 1, and 2 as their values. On the contrary, recency abstraction splits ℓ_0 into two parts: $(\ell_0, \mathbf{o})$ for joined old addresses and $(\ell_0, \mathbf{r})$ for a recent address. At the end of the above code, x has the old abstract address $(\ell_0, \mathbf{o})$ and y has the recent abstract address $(\ell_0, \mathbf{r})$. Thus, x.p has both $\mathbf{undefined}$ and 1 as its values because of weak updates on the old address, but y.p has only 2 as its value because of the strong update on the recent address.

While recency abstraction provides more precise analysis than its underlying address abstraction, it is *not monotone* in the sense that it does not preserve the refinement relationship between its underlying address abstraction techniques. We say that a partition-based address abstraction A_1 with a partition δ_1 is a refinement of another partition-based address abstraction A_2 with a partition δ_2, if the partition δ_1 is finer than the partition δ_2. We prove that the refinement relationship is proportional to the analysis precision. Unfortunately, recency abstraction on A_1, which is a refinement of A_2, may not be a refinement of recency abstraction on A_2. Thus, it is unclear which address abstraction would provide the most precise analysis in conjunction with recency abstraction, which denotes that recency abstraction is *not compositional* with other analysis techniques.

In this paper, we present a *singleton abstraction*, which improves the analysis precision of its underlying address abstraction without the aforementioned problems of recency abstraction. The contributions of this paper include the following:

- We formally define recency abstraction on a partition-based address abstraction such as the allocation-site abstraction, and describes how it interferes with address partitioning and analysis sensitivities.
- We propose a singleton abstraction, which enhances the analysis precision while preserving the refinement relationship of its underlying address abstraction. Therefore, it is compositional with other analysis techniques.
- Our preliminary experimental results show that the singleton abstraction provides similar analysis precision as recency abstraction.

In the remaining of this paper, we present the concrete semantics of a simplified variant of JavaScript (Section 2), formally define recency abstraction, and show two code examples illustrating unintuitive behaviors of recency abstraction (Section 3). Then, we propose a new singleton abstraction, which improves the analysis precision without changing its underlying address abstraction (Section 4). After evaluating the analysis precision of the singleton abstraction compared to recency abstraction (Section 5), we discuss related work (Section 6) and conclude (Section 7).

ℓ_0 : $\mathbf{var}$ $\mathtt{obj} = $ {};
ℓ_1 : $\mathbf{if}$ (?) {
ℓ_2 : $\quad$ $\mathtt{obj.a} = 1$;
ℓ_3 : $\quad$ $\mathtt{obj} = $ {};
ℓ_4 : }

Figure 1. A simple example program

2. Concrete Semantics

In this section, we define the concrete semantics of a simplified variant of JavaScript. It contains essential constructs for address abstraction, and we augment the standard concrete semantics with time information to identify recently created objects. We call such time information a *date*, which is a non-negative integer.

2.1 Notations and Syntax

We use the following notations:

ℓ	$\in$	$\mathbb{L}$	:	control states
$\mathtt{x}, \mathtt{y}$	$\in$	$\mathbb{X}$	:	variables
a	$\in$	$\mathbb{A}$	:	addresses
		$\mathbb{V}_\mathrm{p}$	:	primitive values
		$\mathbb{V}$	:	values ($\mathbb{V} = \mathbb{A} \uplus \mathbb{V}_\mathrm{p}$)
		$\mathbb{D}$	:	dates (non-negative integers)
		$\mathbb{P}$	:	property names of objects (strings)

We let $\odot$ denote the JavaScript undefined value ($\odot \in \mathbb{V}_\mathrm{p}$). A date denotes when an object is created, which is used for recency abstraction. Property names are string values. We consider the following abstract syntax as a simplified variant of JavaScript:

$$
\begin{array}{lll}
\mathit{Program} & ::= & \mathit{Func}^* \ \mathit{Stmt}^* \\
\mathit{Func} & ::= & \mathbf{function}\ \mathit{Id}\ (\ [\mathit{Id}\ [,\ \mathit{Id}]^*]^?\ )\ \{\ \mathit{Stmt}^*\ \} \\
\mathit{Stmt} & ::= & \mathbf{var}\ \mathit{Id}\ [= \mathit{Expr}]^?; \\
& | & \mathit{Id} = \mathit{Expr};\ |\ \mathit{Expr}.\mathit{Prop} = \mathit{Expr}; \\
& | & \mathbf{if}\ (\ \mathit{Expr}\ )\ \{\ \mathit{Stmt}^*\ \}\ \mathbf{else}\ \{\ \mathit{Stmt}^*\ \} \\
& | & \mathbf{return}\ \mathit{Expr}; \\
\mathit{Expr} & ::= & \mathit{Expr}\ (\ \mathit{Expr}^*\ )\ |\ \mathit{Expr}\ [\ \mathit{Expr}\ ]\ \ |\ \mathit{Expr}.\mathit{Prop}\ |\ \{\} \\
& | & \mathit{Id}\ |?\ |\ 0\ |\ 1\ |+|\ -\ |\ \cdots\ \text{(values or operators)} \\
\mathit{Id} & ::= & \mathtt{x}\ |\ \mathtt{y}\ |\ \cdots\ \text{(variable names)} \\
\mathit{Prop} & ::= & \mathtt{a}\ |\ \mathtt{p}\ |\ \cdots\ \text{(property names of objects)}
\end{array}
$$

We use ? to denote unknown values such as dynamically generated values. Figure 1 shows an example code in this syntax. Given a statement s, we write $\ell_0 : \mathtt{s}; \ell_1$, if ℓ_0 is the control state right before the statement and ℓ_1 is the control state right after it.

2.2 States and Traces

A state $\sigma = (\sigma^{\mathbb{L}}, \sigma^{\mathbb{C}}, \sigma^{\mathbb{H}}, \sigma^{\mathbb{D}}) \in \mathbb{S}$ consists of a control state, a context, a heap, and a date:

$\mathbb{S}$	$=$	$\mathbb{L} \times \mathbb{C} \times \mathbb{H} \times \mathbb{D}$	:	states
$\mathbb{C}$	$=$	$\mathbb{E} \times \mathbb{K}$	:	contexts
$\mathbb{E}$	$=$	$\mathbb{X} \rightarrowtail \mathbb{V}$	:	environments
$\mathbb{K}$	$=$	$\{\epsilon\} \uplus (\mathbb{L} \times \mathbb{C})$	:	call contexts
$\mathbb{H}$	$=$	$\mathbb{A} \rightarrowtail (\mathbb{O} \times \mathbb{L} \times \mathbb{D})$	:	heaps
$\mathbb{O}$	$=$	$\mathbb{P} \rightarrowtail \mathbb{V}$	:	objects

A context consists of an environment and a call context. An environment is a partial map from variables to values. A call context in top-level is ϵ; in a function f, a call context is a pair of the control point and the context of the call-site of f. A heap is a partial map from addresses to objects with their allocation sites and dates. An object is a partial map from property names to their values.

A trace $\tau \in \mathbb{T}$ is a finite sequence of states $\langle \sigma_0, \ldots, \sigma_{n-1} \rangle$. The date of a state captures the number of program execution steps so far: in a well-formed trace $\langle \sigma_0, \ldots, \sigma_{n-1} \rangle$, the date of σ_i is i and the transition rules in the concrete semantics defined in Section 2.3 ensure this. The following table represents sample traces for the example code in Figure 1. A trace executing the true branch is as follows:

$\sigma_i^{\mathbb{L}}$	$\sigma_i^{\mathbb{E}}(\mathtt{obj})$	$\sigma_i^{\mathbb{H}}$	$\sigma_i^{\mathbb{D}}$
ℓ_0	$\odot$	$\emptyset$	0
ℓ_1	$a_{\mathtt{t}_0}$	$a_{\mathtt{t}_0} \mapsto (\{\}, \ell_0, 0)$	1
ℓ_2	$a_{\mathtt{t}_0}$	$a_{\mathtt{t}_0} \mapsto (\{\}, \ell_0, 0)$	2
ℓ_3	$a_{\mathtt{t}_0}$	$a_{\mathtt{t}_0} \mapsto (\{\mathtt{a} : 1\}, \ell_0, 0)$	3
ℓ_4	$a_{\mathtt{t}_1}$	$a_{\mathtt{t}_0} \mapsto (\{\mathtt{a} : 1\}, \ell_0, 0)$ $a_{\mathtt{t}_1} \mapsto (\{\}, \ell_3, 3)$	4

and another trace executing the false branch is as follows:

$\sigma_i^{\mathbb{L}}$	$\sigma_i^{\mathbb{E}}(\mathtt{obj})$	$\sigma_i^{\mathbb{H}}$	$\sigma_i^{\mathbb{D}}$
ℓ_0	$\odot$	$\emptyset$	0
ℓ_1	$a_{\mathtt{f}_0}$	$a_{\mathtt{f}_0} \mapsto (\{\}, \ell_0, 0)$	1
ℓ_5	$a_{\mathtt{f}_0}$	$a_{\mathtt{f}_0} \mapsto (\{\}, \ell_0, 0)$	2

2.3 Concrete Semantics

We define a small-step semantics characterized by a transition relation $\rightarrow$, and use the finite trace semantics induced by $\rightarrow$. The initial state is $(\ell_0, (\emptyset, \epsilon), \emptyset, 0)$ where ℓ_0 is the start control point of a given program. The helper function $\mathbf{eval}(e, \sigma^{\mathbb{E}})$ evaluates an expression e with an environment $\sigma^{\mathbb{E}}$. For instance, the transitions for simple variable creation and object allocation statements have the following semantics:

- Simple variable creation without initialization

  $\ell_0 :$ **var** x;
  $\ell_1 :$ ...

  $(\ell_0, (\sigma_0^{\mathbb{E}}, \sigma_0^{\mathbb{K}}), \sigma_0^{\mathbb{H}}, \sigma_0^{\mathbb{D}}) \rightarrow (\ell_1, (\sigma_1^{\mathbb{E}}, \sigma_0^{\mathbb{K}}), \sigma_0^{\mathbb{H}}, \sigma_1^{\mathbb{D}})$ where $\sigma_1^{\mathbb{E}} = \sigma_0^{\mathbb{E}}[\mathtt{x} \mapsto \odot]$ and $\sigma_1^{\mathbb{D}} = \sigma_0^{\mathbb{D}} + 1$
- Object allocation

  $\ell_0 :$ $\mathtt{x} = \{\}$;
  $\ell_1 :$ ...

  $(\ell_0, (\sigma_0^{\mathbb{E}}, \sigma_0^{\mathbb{K}}), \sigma_0^{\mathbb{H}}, \sigma_0^{\mathbb{D}}) \rightarrow (\ell_1, (\sigma_1^{\mathbb{E}}, \sigma_0^{\mathbb{K}}), \sigma_1^{\mathbb{H}}, \sigma_1^{\mathbb{D}})$ where a is a fresh address, $\sigma_1^{\mathbb{E}} = \sigma_0^{\mathbb{E}}[\mathtt{x} \mapsto a]$, $\sigma_1^{\mathbb{H}} = \sigma_0^{\mathbb{H}}[a \mapsto (\{\}, \ell_0, \sigma_0^{\mathbb{D}})]$, and $\sigma_1^{\mathbb{D}} = \sigma_0^{\mathbb{D}} + 1$.

The remaining rules are available in Appendix A. Generally, the transition relation $\rightarrow$ should ensure that, for each transition $(\sigma_0^{\mathbb{L}}, \sigma_0^{\mathbb{C}}, \sigma_0^{\mathbb{H}}, \sigma_0^{\mathbb{D}}) \rightarrow (\sigma_1^{\mathbb{L}}, \sigma_1^{\mathbb{C}}, \sigma_1^{\mathbb{H}}, \sigma_1^{\mathbb{D}})$, $\sigma_1^{\mathbb{D}} = \sigma_0^{\mathbb{D}} + 1$.

3. Recency Abstraction

We formally define a series of abstractions towards recency abstraction on top of a given partition-based address abstraction. Then, we illustrate unintuitive behaviors of recency abstraction using two code examples.

3.1 Abstractions

We define a program abstraction by composing a series of abstractions:

- a classical flow-sensitive abstraction that maps each control state to the set of states that are observed at that location; and
- an abstraction of sets of states by collapsing addresses according to an address abstraction given as a parameter of the program abstraction.

Address abstraction An address abstraction is defined by:

- a set of *abstract addresses* $\mathbb{A}^\sharp$ (we note $a^\sharp$ for an element of this set); and
- a function $\phi^{\mathbb{A}} : \mathbb{A} \longrightarrow \mathbb{A}^\sharp$ that maps each address into the abstract address that represents it.

In the following, we consider several choices of this address abstraction. In each case, it fixes fully for each state the mapping between concrete addresses and abstract addresses.

State abstraction based on address abstraction Given a pair $(\mathbb{A}^\sharp, \phi^{\mathbb{A}})$, we can define abstract domains and abstraction functions for the state abstraction as shown in Figure 2. Abstractions of control states and primitive values are their powersets. Because a power set of primitive values could be an infinite set, we should define its finite abstraction in real analysis, but we use powersets in this paper for the presentation brevity. Abstractions of states are a pair of abstractions of environments and heaps. An abstract environment is a map from variables to pairs of abstract addresses and sets of primitive values. An abstract heap is a map from abstract addresses to abstract objects. In the heap abstraction, we merge all the abstract objects corresponding to a given abstract address. Finally, an abstract object is a map from property names to pairs of abstract addresses and sets of primitive values or the special consider $\circledast$; when an abstract object $o^\sharp$ has a mapping from p to $\circledast$, it denotes that p may not exist in $o^\sharp$. Since abstract domains are complete lattices, we define at each step an element-wise abstraction function ϕ, that maps each element to its best abstraction. Such functions implicitly define Galois connections. For instance, $\mathbb{A}, \phi^{\mathbb{A}}$ defines $\mathcal{P}(\mathbb{A}) \underset{\gamma}{\overset{\alpha}{\rightleftharpoons}} \mathbb{A}^\sharp$ by:

$$\alpha(\mathbb{A}') = \textstyle\bigsqcup_{a \in \mathbb{A}'} \phi^{\mathbb{A}}(a), \quad \gamma(a^\sharp) = \{a \in \mathbb{A} \mid \phi^{\mathbb{A}}(a) \sqsubseteq a^\sharp\}$$

Recency abstraction Recency abstraction is a commonly used address abstraction for JavaScript analysis, which is often defined on top of the allocation-site abstraction. We generalize the allocation-site abstraction as a partition-based address abstraction, and define recency abstraction on it. Thus, our formalization can represent recency abstraction on heap cloning [8], which is yet another partition-based address abstraction.

A partition-based address abstraction $(\mathbb{A}_\delta^\sharp, \phi_\delta^{\mathbb{A}})$ is defined with a partition $\delta : \mathbb{A} \rightarrow \Pi$ where $\mathbb{A}_\delta^\sharp = \mathcal{P}(\Pi)$ and $\phi_\delta^{\mathbb{A}}(a) = \{\delta(a)\}$. We could simplify the heap abstraction using the partition δ as follows:

- $\mathbb{H}_\delta^\sharp = \Pi \longrightarrow \mathbb{O}^\sharp$
- $\phi_\delta^{\mathbb{H}}(h) = \lambda(\pi \in \Pi) \cdot \bigsqcup \{\phi^{\mathbb{O}}(o) \mid \exists a \in \mathbb{A},\ \delta(a) = \pi \wedge h(a) = (o, \_, \_)\}$

Given a partition-based address abstraction with a partition $\delta : \mathbb{A} \rightarrow \Pi$, and a corresponding state (ℓ, c, h, n), we define recency abstraction as follows:

- $\mathbb{A}_{\mathrm{r}[\delta]}^\sharp = \mathcal{P}(\Pi \times \{\mathbf{r}, \mathbf{o}\})$;
- $\phi_{\mathrm{r}[\delta]}^{\mathbb{A}}(a) = \begin{cases} \{(\pi, \mathbf{r})\} & \text{if } h(a) = (o, \ell, n_r) \\ \{(\pi, \mathbf{o})\} & \text{otherwise} \end{cases}$

Concrete domain	Abstract domain	Element-wise abstraction function
$\mathbb{S}$	$\mathbb{S}^\sharp = \mathbb{E}^\sharp \times \mathbb{H}^\sharp$	$\phi^{\mathbb{S}}((\_, (e, \_), h, \_)) = (\phi^{\mathbb{E}}(e), \phi^{\mathbb{H}}(h))$
$\mathbb{E}$	$\mathbb{E}^\sharp = \mathbb{X} \longrightarrow \mathbb{A}^\sharp \times \mathcal{P}(\mathbb{V}_{\mathrm{p}})$	$\phi^{\mathbb{E}}(e) = \lambda(\mathtt{x} \in \mathit{Domain}(e)) \cdot \begin{cases} (\phi^{\mathbb{A}}(e(\mathtt{x})), \{\}) & \text{if } e(\mathtt{x}) \text{ is an address} \\ (\bot, \{e(\mathtt{x})\}) & \text{if } e(\mathtt{x}) \text{ is a primitive value} \end{cases}$
$\mathbb{H}$	$\mathbb{H}^\sharp = \mathbb{A}^\sharp \longrightarrow \mathbb{O}^\sharp$	$\phi^{\mathbb{H}}(h) = \lambda(a^\sharp \in \mathbb{A}^\sharp) \cdot \bigsqcup \{\phi^{\mathbb{O}}(o) \mid \exists a \in \mathbb{A},\ \phi^{\mathbb{A}}(a) = a^\sharp \wedge h(a) = (o, \_, \_)\}$
$\mathbb{O}$	$\mathbb{O}^\sharp = \mathbb{P} \longrightarrow \mathbb{A}^\sharp \times \mathcal{P}(\mathbb{V}_{\mathrm{p}} \uplus \{\circledast\})$	$\phi^{\mathbb{O}}(o) = \lambda(\mathtt{p}) \cdot \begin{cases} (\phi^{\mathbb{A}}(o(\mathtt{p})), \{\}) & \text{if } o(\mathtt{p}) \text{ is an address} \\ (\bot, \{o(\mathtt{p})\}) & \text{if } o(\mathtt{p}) \text{ is a primitive value} \\ (\bot, \{\circledast\}) & \text{if } \mathtt{p} \notin \mathit{Domain}(o) \end{cases}$

Figure 2. Abstractions based on the address abstraction $(\mathbb{A}^\sharp, \phi^{\mathbb{A}})$

where $\pi = \delta(a)$
and $\quad n_r = \max\{n' \mid \exists a' \in \mathbb{A}, o' \in \mathbb{O}, \ell' \in \mathbb{L},$
$$\delta(a') = \pi \wedge h(a') = (o', \ell', n')\}.$$
This allows to abstract sets of states similarly as above except that the address abstraction function depends on states.

3.2 Unintuitive Behaviors of Recency Abstraction

Now, we illustrate unintuitive behaviors of recency abstraction using two examples.

Example 1 The code in Figure 1 contains two allocation-sites ℓ_0 and ℓ_3. Let us consider two partition-based address abstractions: the allocation-site abstraction with $\delta_{id} : \mathbb{A} \to \mathbb{L}$, which divides addresses based on their allocation sites and a crude one with $\delta_\top : \mathbb{A} \to \{\pi\}$ for some π, which does not partition at all. Clearly, the abstraction $(\mathbb{A}^\sharp_{\delta_{id}}, \phi^{\mathbb{A}}_{\delta_{id}})$ defines a more precise address partition than $(\mathbb{A}^\sharp_{\delta_\top}, \phi^{\mathbb{A}}_{\delta_\top})$. Unfortunately, recency abstraction does not preserve this "more precise than" relationship. Let's look at the analysis results at the control state ℓ_4. The abstraction with recency abstraction $(\mathbb{A}^\sharp_{\mathrm{r}[\delta_{\mathrm{id}}]}, \phi^{\mathbb{A}}_{\mathrm{r}[\delta_{\mathrm{id}}]})$ produces the following result:

	$e^\sharp$	$h^\sharp$
true branch	$\mathtt{obj} \mapsto \{(\ell_3, \mathbf{r})\}$	$(\ell_0, \mathbf{r}) \mapsto \{\mathtt{a} \mapsto \{1\}\}$ $(\ell_3, \mathbf{r}) \mapsto \{\}$
false branch	$\mathtt{obj} \mapsto \{(\ell_0, \mathbf{r})\}$	$(\ell_0, \mathbf{r}) \mapsto \{\}$
join	$\mathtt{obj} \mapsto \{(\ell_0, \mathbf{r}),$ $(\ell_3, \mathbf{r})\}$	$(\ell_0, \mathbf{r}) \mapsto \{\mathtt{a} \mapsto \{\circledast, 1\}\}$ $(\ell_3, \mathbf{r}) \mapsto \{\}$

The joined result of both true and false branches shows that `obj.a` may have values $\{\circledast, 1\}$. On the contrary, the abstraction with $(\mathbb{A}^\sharp_{\mathrm{r}[\delta_\top]}, \phi^{\mathbb{A}}_{\mathrm{r}[\delta_\top]})$ produces the following:

	$e^\sharp$	$h^\sharp$
true branch	$\mathtt{obj} \mapsto \{(\pi, \mathbf{r})\}$	$(\pi, \mathbf{r}) \mapsto \{\}$ $(\pi, \mathbf{o}) \mapsto \{\mathtt{a} \mapsto \{1\}\}$
false branch	$\mathtt{obj} \mapsto \{(\pi, \mathbf{r})\}$	$(\pi, \mathbf{r}) \mapsto \{\}$
join	$\mathtt{obj} \mapsto \{(\pi, \mathbf{r})\}$	$(\pi, \mathbf{r}) \mapsto \{\}$ $(\pi, \mathbf{o}) \mapsto \{\mathtt{a} \mapsto \{1\}\}$

The joined result of both branches shows that `a` does not exist in `obj`; thus, the value of `obj.a` is $\{\circledast\}$. This example shows that $(\mathbb{A}^\sharp_{\mathrm{r}[\delta_\top]}, \phi^{\mathbb{A}}_{\mathrm{r}[\delta_\top]})$ is more precise than $(\mathbb{A}^\sharp_{\mathrm{r}[\delta_{\mathrm{id}}]}, \phi^{\mathbb{A}}_{\mathrm{r}[\delta_{\mathrm{id}}]})$ while $(\mathbb{A}^\sharp_{\delta_{id}}, \phi^{\mathbb{A}}_{\delta_{id}})$ is more precise than $(\mathbb{A}^\sharp_{\delta_\top}, \phi^{\mathbb{A}}_{\delta_\top})$. Therefore, the precision relationship of the underlying address abstraction is not preserved with recency abstraction.

Example 2 The code in Figure 3 shows that recency abstraction may interfere with analysis sensitivities. Let us consider two

```
ℓ0 :   function g(z){
ℓ1 :       var result = z.p;
ℓ2 :   }
ℓ3 :   function f(){
ℓ4 :       var obj = {};
ℓ5 :       var a = g(obj);
ℓ6 :       obj.p = 3;
ℓ7 :       return obj;
ℓ8 :   }
ℓ9 :   var x = f();
ℓ10 :  var y = f();
ℓ11 :
```

Figure 3. Recency abstraction interfering with sensitivities

analysis sensitivities: 1-CFA that distinguishes the same function from its different call sites using its caller, and 0-CFA that does not distinguish different call sites of the same function. Then, we consider the allocation-site abstraction refined by different sensitivities. With 1-CFA, the partition is $\delta : \mathbb{A} \to \{\ell_{4/9}, \ell_{4/10}\}$ where $\ell_{4/9}$ means that the allocation-site ℓ_4 with the call-site ℓ_9 and $\ell_{4/10}$ means that the allocation-site ℓ_4 with the call-site ℓ_{10}. In this case, we get the following result at the control state ℓ_1:

	$e^\sharp$	$h^\sharp$
call ℓ_9, ℓ_5	$\mathtt{z} \mapsto \{(\ell_{4/9}, \mathbf{r})\}$	$(\ell_{4/9}, \mathbf{r}) \mapsto \{\}$
call ℓ_{10}, ℓ_5	$\mathtt{z} \mapsto \{(\ell_{4/10}, \mathbf{r})\}$	$(\ell_{4/9}, \mathbf{r}) \mapsto \{\mathtt{p} \mapsto \{3\}\}$ $(\ell_{4/10}, \mathbf{r}) \mapsto \{\}$
join	$\mathtt{z} \mapsto \{(\ell_{4/9}, \mathbf{r}),$ $(\ell_{4/10}, \mathbf{r})\}$	$(\ell_{4/9}, \mathbf{r}) \mapsto \{\mathtt{p} \mapsto \{\circledast, 3\}\}$ $(\ell_{4/10}, \mathbf{r}) \mapsto \{\}$

With 0-CFA, the partition is $\delta : \mathbb{A} \to \{\ell_4\}$. Thus, it has only one partition and we get the following result at the control state ℓ_1:

	$e^\sharp$	$h^\sharp$
call ℓ_9, ℓ_5	$\mathtt{z} \mapsto \{(\ell_4, \mathbf{r})\}$	$(\ell_4, \mathbf{r}) \mapsto \{\}$
callℓ_{10}, ℓ_5	$\mathtt{z} \mapsto \{(\ell_4, \mathbf{r})\}$	$(\ell_4, \mathbf{r}) \mapsto \{\}$ $(\ell_4, \mathbf{o}) \mapsto \{\mathtt{p} \mapsto \{3\}\}$
join	$\mathtt{z} \mapsto \{(\ell_4, \mathbf{r})\}$	$(\ell_4, \mathbf{r}) \mapsto \{\}$ $(\ell_4, \mathbf{o}) \mapsto \{\mathtt{p} \mapsto \{3\}\}$

This example shows that a more precise 1-CFA may produce less precise results than 0-CFA when combined with recency abstraction. Therefore, the precision relationship of analysis sensitivities is not preserved when combined with recency abstraction.

4. Singleton Abstraction

In this section, we explain the unintuitive behaviors of recency abstraction in terms of the refinement relationship between partition-based address abstractions. Then, we present *singleton abstraction*, a new heap abstraction based on a given partition-based address abstraction, which preserves the refinement relationship of its underlying address abstraction and moreover allows strong updates on singleton addresses.

4.1 Refinement of Address Abstraction

We first define terminologies to discuss the behaviors of recency abstraction. A partition-based address abstractions $(\mathbb{A}^\sharp_{\delta_i}, \phi^{\mathbb{A}}_{\delta_i})$ is defined with a partition $\delta_i : \mathbb{A} \rightarrow \Pi_i$. A partition-based address abstraction is a refinement of another, if and only if their partitions have the refinement relationship accordingly.

Definition 1 ($\preceq$)**.** $(\mathbb{A}^\sharp_{\delta_1}, \phi^{\mathbb{A}}_{\delta_1}) \preceq (\mathbb{A}^\sharp_{\delta_2}, \phi^{\mathbb{A}}_{\delta_2})$ *iff* δ_1 *is a refinement partition of* δ_2*.*

An address abstraction $(\mathbb{A}^\sharp_1, \phi^{\mathbb{A}}_1)$ is more precise than $(\mathbb{A}^\sharp_2, \phi^{\mathbb{A}}_2)$ if and only if the concretization of the former is a subset of that of the latter.

Definition 2 ($\preceq_p$)**.** $(\mathbb{A}^\sharp_1, \phi^{\mathbb{A}}_1) \preceq_p (\mathbb{A}^\sharp_2, \phi^{\mathbb{A}}_2)$ *iff* $\gamma_1 \circ \alpha_1 \subseteq \gamma_2 \circ \alpha_2$*.*

Then, we prove that the refinement relation implies the precision relation.

Theorem 1 (Implication of precision from refinement)**.**

$$(\mathbb{A}^\sharp_{\delta_1}, \phi^{\mathbb{A}}_{\delta_1}) \preceq (\mathbb{A}^\sharp_{\delta_2}, \phi^{\mathbb{A}}_{\delta_2}) \Rightarrow (\mathbb{A}^\sharp_{\delta_1}, \phi^{\mathbb{A}}_{\delta_1}) \preceq_p (\mathbb{A}^\sharp_{\delta_2}, \phi^{\mathbb{A}}_{\delta_2})$$

Proof. Let us consider the Galois connections $\mathcal{P}(\mathbb{A}) \underset{\gamma_1}{\overset{\alpha_1}{\rightleftarrows}} \mathbb{A}^\sharp_{\delta_1}$ and $\mathcal{P}(\mathbb{A}) \underset{\gamma_2}{\overset{\alpha_2}{\rightleftarrows}} \mathbb{A}^\sharp_{\delta_2}$ defined as above. Given $\mathbb{A}' \subseteq \mathbb{A}$, there exists $\Pi'_1 \subseteq \Pi_1$ s.t. $\gamma_1(\Pi'_1) = \gamma_2(\Pi'_2)$ where $\Pi'_2 = \alpha_2(\mathbb{A}')$ because $(\mathbb{A}^\sharp_{\delta_1}, \phi^{\mathbb{A}}_{\delta_1}) \preceq (\mathbb{A}^\sharp_{\delta_2}, \phi^{\mathbb{A}}_{\delta_2})$. It means that $\alpha_1(\mathbb{A}') \sqsubseteq \Pi'_1$. Therefore, $\gamma_1 \circ \alpha_1(\mathbb{A}') \subseteq \gamma_1(\Pi'_1) = \gamma_2(\Pi'_2) = \gamma_2 \circ \alpha_2(\mathbb{A}')$. □

Now, let us revisit the first example in Section 3.2 with the refinement relation. Because δ_{id} is a partition of $\delta_\top$, we have $(\mathbb{A}^\sharp_{\delta_{id}}, \phi^{\mathbb{A}}_{\delta_{id}}) \preceq (\mathbb{A}^\sharp_{\delta_\top}, \phi^{\mathbb{A}}_{\delta_\top})$. The recency abstraction with a cruder partition $(\mathbb{A}^\sharp_{\mathrm{r}[\delta_\top]}, \phi^{\mathbb{A}}_{\mathrm{r}[\delta_\top]})$ has two partitions $(\pi, \mathbf{r})$ and $(\pi, \mathbf{o})$: $\gamma((\pi, \mathbf{r})) = \{a_{\mathbf{t}_1}, a_{\mathbf{f}_0}\}$ and $\gamma((\pi, \mathbf{o})) = \{a_{\mathbf{t}_0}\}$ where $a_{\mathbf{t}_0}$ and $a_{\mathbf{t}_1}$ are concrete addresses created at ℓ_0 and ℓ_3, respectively, for the true branch, and $a_{\mathbf{f}_0}$ is a concrete address created at ℓ_0 for the false branch. The other recency abstraction $(\mathbb{A}^\sharp_{\mathrm{r}[\delta_{\mathrm{id}}]}, \phi^{\mathbb{A}}_{\mathrm{r}[\delta_{\mathrm{id}}]})$ has four partitions, and only two partitions $(\ell_0, \mathbf{r})$ and $(\ell_3, \mathbf{r})$ have elements: $\gamma((\ell_0, \mathbf{r})) = \{a_{\mathbf{t}_0}, a_{\mathbf{f}_0}\}$ and $\gamma((\ell_3, \mathbf{r})) = \{a_{\mathbf{t}_1}\}$. Thus, $(\mathbb{A}^\sharp_{\mathrm{r}[\delta_{\mathrm{id}}]}, \phi^{\mathbb{A}}_{\mathrm{r}[\delta_{\mathrm{id}}]}) \not\preceq (\mathbb{A}^\sharp_{\mathrm{r}[\delta_\top]}, \phi^{\mathbb{A}}_{\mathrm{r}[\delta_\top]})$, which illustrates a case where recency abstraction does not preserve the refinement relation of its underlying address abstraction, which in turn does not preserve their precision relation. Similarly, the second example shows that recency abstraction with a more precise 1-CFA analysis sensitivity does not always produce more precise analysis results than recency abstraction with a less precise 0-CFA.

4.2 Singleton Abstraction

To alleviate the problem, we decide not to divide a given partition but to simply perform strong updates on singleton objects. Thus, we propose singleton abstraction, a new heap abstraction that preserves the refinement relationship of its underlying address abstraction. It can provide more precise analysis results

Bench	Program	LOC	Recency	Singleton	Total
JSAI	adn-chess.js	234	90	55	127
	adn-coffee_pods_deals.js	367	45	37	141
	adn-less_spam_please.js	759	213	143	432
	adn-live_pagerank.js	882	132	117	323
	adn-odesk_job_watcher.js	168	56	52	71
	adn-pinpoints.js	548	58	57	232
	adn-tryagain.js	929	103	72	525
SunSpider	3d-morph.js	23	1	1	4
	access-binary-trees.js	38	14	10	16
	access-fannkuch.js	51	1	1	19
	access-nbody.js	142	32	15	78
	access-nsieve.js	28	2	0	4
	bitops-3bit-bits-in-byte.js	13	0	0	0
	bitops-bits-in-byte.js	14	0	0	0
	bitops-bitwise-and.js	3	0	0	0
	bitops-nsieve-bits.js	22	1	1	7
	controlflow-recursive.js	18	0	0	0
	math-cordic.js	53	4	4	6
	math-partial-sums.js	25	4	4	4
	math-spectral-norm.js	41	2	1	16
	string-fasta.js	70	15	10	18
V8	navier-stokes.js	331	36	17	92
	richards.js	288	119	117	197
	splay.js	205	108	108	132
Total			1036	831	2,444
Ratio (%)			42.39	33.63	–

Table 1. Numbers of object property loads that have more precise results with recency or singleton abstraction than the allocation-site abstraction.

with more precise underlying address abstractions and with more precise analysis sensitivities.

Given a partition-based address abstraction $(\mathbb{A}^\sharp_\delta, \phi^{\mathbb{A}}_\delta)$ with a partition $\delta = \mathbb{A} \rightarrow \Pi$, we define singleton abstraction as follows:

- $\mathbb{H}^\sharp_{\mathrm{s}[\delta]} = \Pi \longrightarrow \mathbb{O}^\sharp \times \{\mathbf{s}, \mathbf{m}\}$;
- $\phi^{\mathbb{H}}_{\mathrm{s}[\delta]}(h) = \lambda(\pi \in \Pi) \cdot (\phi^{\mathbb{H}}_\delta(\pi), \begin{cases} \mathbf{s} & \text{if } |U| = 1 \\ \mathbf{m} & \text{otherwise} \end{cases})$

  where $U = \{a' \in \mathbb{A} \mid \delta(a') = \pi \wedge a' \in \mathit{Domain}(h)\}$.

It distinguishes partitions with only one address as $\mathbf{s}$ and maps the other partitions to $\mathbf{m}$. Merging two mappings from the same partition to both singleton $(\mathbf{s})$ and multiple $(\mathbf{m})$ results in $\mathbf{m}$.

Unlike recency abstraction, the singleton abstraction preserves the refinement relation of its underlying address abstraction because they use the same partition from the underlying address abstraction. While the expressive power of the singleton abstraction is the same as its partition-based address abstraction, singleton abstraction allows strong updates for address partitions that map to $\mathbf{s}$. It permits strong updates on objects created at specific allocation sites.

5. Evaluation

We evaluate the precision of singleton abstraction in comparison with recency abstraction. We conducted experiments with 3 sets of benchmarks—JSAI, SunSpider, and V8—consisting of 24 programs on a 2.8 GHz Intel Core i5 iMac with 16GB memory. We implemented 3 address abstractions—allocation-site abstraction, recency abstraction, and singleton abstraction—on an opensource JavaScript static analysis framework, SAFE [9]. The implemented recency abstraction and singleton abstraction are built on top of the allocation-site abstraction.

The analyses took on average 86.92, 122.73, and 79.77 seconds for the allocation-site, recency, and singleton abstractions, respectively. It means that singleton abstraction does not incur much performance overhead like recency abstraction while providing comparable analysis precision with recency abstraction. We observed that the more complex benchmark programs get, the more performance overhead recency abstraction causes.

For the analysis precision, we compare the numbers of object property loads like `obj.p` that have more precise results with recency or singleton abstraction compared with just the allocation-site abstraction. Table 1 summarizes the experimental results; the 3rd column shows the lines of code, the 4th and the 5th columns show the numbers of more precise property loads by recency and singleton abstractions, respectively, and the last column shows the total number of property loads in each program. For example, the first program in the JSAI benchmarks, adn-chess.js, has 127 property loads, among which recency abstraction analzyes 90 property loads more precisely than the allocation-site abstraction. In summary, recency and singleton abstractions analyze about 42.39% and 33.63% of property loads more precisely on average, respectively. Note that recency abstraction divides partitions into two parts: recent and old. Therefore, recency abstraction provides more precise analysis results than singleton abstraction when programs update recent addresses and their allocation sites also have old addresses pointing to different shapes of objects. We plan to extend the set of benchmark programs to understand the relationships between recency and singleton abstractions more clearly.

6. Related Work

Among various research directions on heap abstraction, store-based heap models are the abstractions that represent abstract addresses as graph nodes [7]. The allocation-site abstraction is the most commonly used store-based heap model and recency abstraction is also store-based. In addition, our singleton abstraction is also a store-based heap model.

Recency abstraction was first proposed by Balakrishnan *et al.* [4] in 2006 to resolve virtual function calls in C++ by supporting strong updates. While existing approaches allow only weak updates on abstract malloc blocks because they may represent summary blocks containing multiple blocks, recency abstraction permits strong updates for malloc blocks, which could resolve 55% of virtual function calls in their experiments. However, they use only the allocation-site address abstraction as the underlying address abstraction for recency. On the contrary, we generalized the underlying address abstraction to partition-based address abstraction that includes the allocation-site abstraction.

Heidegger *et al.* applied recency abstraction for JavaScript analysis [5]. While they captured recency information in a type system and developed an inference algorithm using a constraint solver for it, we defined recency abstraction entirely based on the abstract interpretation framework.

Most JavaScript static analyzers now use recency abstraction based on the allocation-site abstraction [6, 9], and they provide an option to use heap cloning [8]. The heap cloned allocation-site abstraction is also a partition-based address abstraction, which is a refinement of the allocation-site abstraction. Because we defined recency abstraction on top of a partition-based address abstraction, our formalization of the recency abstraction covers the recency abstraction implementation in real-world JavaScript analyzers.

7. Conclusion

We revisited recency abstraction, a typical address abstraction technique for static analysis of JavaScript programs. We formally defined it on a partition-based address abstraction, and we used the formalization to describe unintuitive behaviors of recency abstraction. We explained the behaviors by showing that recency abstraction does not preserve the refinement relationship between its underlying address abstractions. Thus, it is difficult to predict which address abstraction would provide the most precise analysis result for recency abstraction.

Thus, we proposed singleton abstraction, a new heap abstraction using a partition-based abstraction. It preserves the refinement relationship of the underlying address abstractions. Therefore, it is compositional with other analysis techniques. Moreover, our preliminary experiments showed that it provides similar analysis precision with recency abstraction while reducing the performance overhead.

Acknowledgment. The research leading to these results has received funding from the European Research Council under the EU FP 7, grant Agreement 278673, Project MemCAD.

References

[1] TIOBE Index for February 2017. `http://www.tiobe.com/tiobe-index`.

[2] Iot.js: A framework for Internet of Things. `http://samsung.github.io/jerryscript/`, 2015.

[3] E. Andreasen and A. Møller. Determinacy in static analysis for jQuery. In *OOPSLA*, 2014.

[4] G. Balakrishnan and T. Reps. Recency-abstraction for heap-allocated storage. In *SAS*, 2006.

[5] P. Heidegger and P. Thiemann. Recency types for analyzing scripting languages. In *ECOOP*, 2010.

[6] S. H. Jensen, A. Møller, and P. Thiemann. Type analysis for JavaScript. In *SAS*, 2009.

[7] V. Kanvar and U. P. Khedker. Heap abstractions for static analysis. In *ACM Computing Surveys (CSUR)*, 2016.

[8] C. Lattner, A. Lenharth, and V. Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *PLDI*, 2007.

[9] H. Lee, S. Won, J. Jin, J. Cho, and S. Ryu. SAFE: Formal specification and implementation of a scalable analysis framework for ECMAScript. In *FOOL*, 2012.

[10] C. Park and S. Ryu. Scalable and precise static analysis of JavaScript applications via loop-sensitivity. In *ECOOP*, 2015.

[11] M. Schäfer, M. Sridharan, J. Dolby, and F. Tip. Dynamic determinacy analysis. In *PLDI*, 2013.

[12] M. Sridharan, J. Dolby, S. Chandra, M. Schäfer, and F. Tip. Correlation tracking for points-to analysis of JavaScript. In *ECOOP*, 2012.

A. Semantics of "$\rightarrow$"

We define the semantics of $\rightarrow$ for the simplified JavaScript. For each statement or expression that changes a state, we provide its state transition rule. In the rules, the special variable RET initialized to the undefined value $\odot$ denotes return values. The helper function $\mathbf{eval}(\mathtt{e}, \sigma)$ evaluates an expression e with a state σ. We write σ_i to denote the state at the control state ℓ_i.

1. Simple variable creation without initialization

$$\begin{array}{ll} \ell_0: & \mathbf{var}\ \mathtt{x}; \\ \ell_1: & \ldots \end{array}$$

$(\ell_0, (\sigma_0^{\mathbb{E}}, \sigma_0^{\mathbb{K}}), \sigma_0^{\mathbb{H}}, \sigma_0^{\mathbb{D}}) \rightarrow (\ell_1, (\sigma_1^{\mathbb{E}}, \sigma_0^{\mathbb{K}}), \sigma_0^{\mathbb{H}}, \sigma_1^{\mathbb{D}})$ where

$$\begin{array}{rcl} \sigma_1^{\mathbb{E}} & = & \sigma_0^{\mathbb{E}}[\mathtt{x} \mapsto \odot] \\ \sigma_1^{\mathbb{D}} & = & \sigma_0^{\mathbb{D}} + 1 \end{array}$$

2. Variable creation with initialization

$$\begin{array}{ll} \ell_0: & \mathbf{var}\ \mathtt{x} = \mathtt{e}; \\ \ell_1: & \ldots \end{array}$$

$(\ell_0, (\sigma_0^{\mathbb{E}}, \sigma_0^{\mathbb{K}}), \sigma_0^{\mathbb{H}}, \sigma_0^{\mathbb{D}}) \rightarrow (\ell_1, (\sigma_1^{\mathbb{E}}, \sigma_0^{\mathbb{K}}), \sigma_0^{\mathbb{H}}, \sigma_1^{\mathbb{D}})$ where

$$\begin{array}{rcl} \mathbf{eval}(\mathtt{e}, \sigma_0) & = & v \\ \sigma_1^{\mathbb{E}} & = & \sigma_0^{\mathbb{E}}[\mathtt{x} \mapsto v] \\ \sigma_1^{\mathbb{D}} & = & \sigma_0^{\mathbb{D}} + 1 \end{array}$$

3. Function call statement

$$\begin{array}{ll} \ell_0: & \mathbf{function}\ \mathtt{f}(\mathtt{p}_0, \ldots, \mathtt{p}_{n-1})\{ \\ \ell_1: & \quad\ldots \\ \ell_2: & \} \\ \ell_3: & \mathtt{x} = \mathtt{f}(\mathtt{e}_0, \ldots, \mathtt{e}_{n-1}); \\ \ell_4: & \ldots \end{array}$$

$(\ell_3, (\sigma_3^{\mathbb{E}}, \sigma_3^{\mathbb{K}}), \sigma_3^{\mathbb{H}}, \sigma_3^{\mathbb{D}}) \rightarrow (\ell_1, (\sigma_1^{\mathbb{E}}, \sigma_1^{\mathbb{K}}), \sigma_3^{\mathbb{H}}, \sigma_1^{\mathbb{D}})$ where

$$\begin{array}{rcl} \mathbf{eval}(\mathtt{e}_i, \sigma_3) & = & v_i \\ \sigma_1^{\mathbb{E}} & = & \{\mathtt{RET} \mapsto \odot, \mathtt{p}_i \mapsto v_i\} \\ \sigma_1^{\mathbb{K}} & = & (\ell_3, (\sigma_3^{\mathbb{E}}, \sigma_3^{\mathbb{K}})) \\ \sigma_1^{\mathbb{D}} & = & \sigma_3^{\mathbb{D}} + 1 \end{array}$$

$(\ell_2, (\sigma_2^{\mathbb{E}}, \sigma_2^{\mathbb{K}}), \sigma_2^{\mathbb{H}}, \sigma_2^{\mathbb{D}}) \rightarrow (\ell_4, (\sigma_4^{\mathbb{E}}, \sigma_4^{\mathbb{K}}), \sigma_2^{\mathbb{H}}, \sigma_4^{\mathbb{D}})$ where $\sigma_2^{\mathbb{K}} = (\_, \_, (e, k)), \sigma_2^{\mathbb{E}}(\mathtt{RET}) = v$, and:

$$\begin{array}{rcl} \sigma_4^{\mathbb{E}} & = & e[\mathtt{x} \mapsto v] \\ \sigma_4^{\mathbb{K}} & = & k \\ \sigma_4^{\mathbb{D}} & = & \sigma_2^{\mathbb{D}} + 1 \end{array}$$

4. Allocation statement

$$\begin{array}{ll} \ell_0: & \mathtt{x} = \{\}; \\ \ell_1: & \ldots \end{array}$$

$(\ell_0, (\sigma_0^{\mathbb{E}}, \sigma_0^{\mathbb{K}}), \sigma_0^{\mathbb{H}}, \sigma_0^{\mathbb{D}}) \rightarrow (\ell_1, (\sigma_1^{\mathbb{E}}, \sigma_0^{\mathbb{K}}), \sigma_1^{\mathbb{H}}, \sigma_1^{\mathbb{D}})$ where a is a fresh address, and:

$$\begin{array}{rcl} \sigma_1^{\mathbb{E}} & = & \sigma_0^{\mathbb{E}}[\mathtt{x} \mapsto a] \\ \sigma_1^{\mathbb{H}} & = & \sigma_0^{\mathbb{H}}[a \mapsto (\{\}, \ell_0, \sigma_0^{\mathbb{D}})] \\ \sigma_1^{\mathbb{D}} & = & \sigma_0^{\mathbb{D}} + 1 \end{array}$$

5. Assignment statement

$$\begin{array}{ll} \ell_0: & \mathtt{x} = \mathtt{e}; \\ \ell_1: & \ldots \end{array}$$

$(\ell_0, (\sigma_0^{\mathbb{E}}, \sigma_0^{\mathbb{K}}), \sigma_0^{\mathbb{H}}, \sigma_0^{\mathbb{D}}) \rightarrow (\ell_1, (\sigma_1^{\mathbb{E}}, \sigma_0^{\mathbb{K}}), \sigma_0^{\mathbb{H}}, \sigma_1^{\mathbb{D}})$ where

$$\begin{array}{rcl} \mathbf{eval}(\mathtt{e}, \sigma_0) & = & v \\ \sigma_1^{\mathbb{E}} & = & \sigma_0^{\mathbb{E}}[\mathtt{x} \mapsto v] \\ \sigma_1^{\mathbb{D}} & = & \sigma_0^{\mathbb{D}} + 1 \end{array}$$

6. Property store statement

$$\begin{array}{ll} \ell_0: & \mathtt{x.p} = \mathtt{e}; \\ \ell_1: & \ldots \end{array}$$

$(\ell_0, (\sigma_0^{\mathbb{E}}, \sigma_0^{\mathbb{K}}), \sigma_0^{\mathbb{H}}, \sigma_0^{\mathbb{D}}) \rightarrow (\ell_1, (\sigma_0^{\mathbb{E}}, \sigma_0^{\mathbb{K}}), \sigma_1^{\mathbb{H}}, \sigma_1^{\mathbb{D}})$ where $\sigma_0^{\mathbb{E}}(\mathtt{x}) = a,\ \sigma_0^{\mathbb{H}}(a) = o,\ \mathbf{eval}(\mathtt{e}, \sigma_0) = v$, and:

$$\begin{array}{rcl} \sigma_1^{\mathbb{H}} & = & \sigma_0^{\mathbb{H}}[a \mapsto o[\mathtt{p} \mapsto v]] \\ \sigma_1^{\mathbb{D}} & = & \sigma_0^{\mathbb{D}} + 1 \end{array}$$

7. Branch statement

$$\begin{array}{ll} \ell_0: & \mathbf{if}\ (\ \mathtt{e}\ )\ \{ \\ \ell_1: & \quad\ldots \\ \ell_2: & \}\ \mathbf{else}\ \{ \\ \ell_3: & \quad\ldots \\ \ell_4: & \} \end{array}$$

If $\mathbf{eval}(\mathtt{e}, \sigma_0)$ is true,
$(\ell_0, (\sigma_0^{\mathbb{E}}, \sigma_0^{\mathbb{K}}), \sigma_0^{\mathbb{H}}, \sigma_0^{\mathbb{D}}) \rightarrow (\ell_1, (\sigma_0^{\mathbb{E}}, \sigma_0^{\mathbb{K}}), \sigma_0^{\mathbb{H}}, \sigma_1^{\mathbb{D}})$ where

$$\sigma_1^{\mathbb{D}} = \sigma_0^{\mathbb{D}} + 1$$

and $(\ell_2, (\sigma_2^{\mathbb{E}}, \sigma_2^{\mathbb{K}}), \sigma_2^{\mathbb{H}}, \sigma_2^{\mathbb{D}}) \rightarrow (\ell_4, (\sigma_2^{\mathbb{E}}, \sigma_2^{\mathbb{K}}), \sigma_2^{\mathbb{H}}, \sigma_4^{\mathbb{D}})$ where

$$\sigma_4^{\mathbb{D}} = \sigma_2^{\mathbb{D}} + 1$$

Otherwise, $(\ell_0, (\sigma_0^{\mathbb{E}}, \sigma_0^{\mathbb{K}}), \sigma_0^{\mathbb{H}}, \sigma_0^{\mathbb{D}}) \rightarrow (\ell_3, (\sigma_0^{\mathbb{E}}, \sigma_0^{\mathbb{K}}), \sigma_0^{\mathbb{H}}, \sigma_3^{\mathbb{D}})$ where

$$\sigma_3^{\mathbb{D}} = \sigma_0^{\mathbb{D}} + 1$$

8. Return statement

$$\begin{array}{ll} \ell_0: & \mathbf{return}\ \mathtt{e}; \\ \ell_1: & \ldots \end{array}$$

$(\ell_0, (\sigma_0^{\mathbb{E}}, \sigma_0^{\mathbb{K}}), \sigma_0^{\mathbb{H}}, \sigma_0^{\mathbb{D}}) \rightarrow (\ell_1, (\sigma_1^{\mathbb{E}}, \sigma_0^{\mathbb{K}}), \sigma_0^{\mathbb{H}}, \sigma_1^{\mathbb{D}})$ where

$$\begin{array}{rcl} \mathbf{eval}(\mathtt{e}, \sigma_0) & = & v \\ \sigma_1^{\mathbb{E}} & = & \sigma_0^{\mathbb{E}}[\mathtt{RET} \mapsto v] \\ \sigma_1^{\mathbb{D}} & = & \sigma_0^{\mathbb{D}} + 1 \end{array}$$