A Meta-level Static Analysis for JavaScript

Jihyeok Park
Oracle Labs
Brisbane, Australia
jihyeok.park@oracle.com

Seungmin An
KAIST
Daejeon, South Korea
h2oche@kaist.ac.kr

Sukyoung Ryu
School of Computing
KAIST
Daejeon, South Korea
sryu.cs@kaist.ac.kr

In this report, we formalize a *meta-level static analysis* for JavaScript as a *defined*-language with IR_{ES} as a *defining*-language. We first define IR_{ES} and a JavaScript *definitional interpreter* as an IR_{ES} program. Then, we define a meta-level static analysis for JavaScript with the abstract semantics of IR_{ES} in the abstract interpretation framework [2, 3]. In addition, we explain how to indirectly express abstract domains and analysis sensitivities for JavaScript.

1 IR_{ES}: An IR for ECMAScript Specification

We first define IR_{ES}, an Intermediate Representation for ECMAScript, with its collecting and restricted semantics.

$$
\begin{array}{lll}
\text{Programs} & \mathfrak{P} \ni P ::= f^* \\
\text{Functions} & \mathcal{F} \ni f ::= \texttt{syntax}^? \; \texttt{def} \; \texttt{x}(\texttt{x}^*) \; \{[\ell : i]^*\} \\
\text{Variables} & \mathcal{X} \ni \texttt{x} \\
\text{Labels} & \mathcal{L} \ni \ell \\
\text{Instructions} & \mathcal{I} \ni i ::= r := e \mid \texttt{x} := \{\} \mid \texttt{x} := e(e^*) \\
& \qquad\quad \mid \texttt{if} \; e \; \ell \; \ell \mid \texttt{return} \; e \\
\text{Expressions} & \mathcal{E} \ni e ::= v^{\text{p}} \mid \text{op}(e^*) \mid r \\
\text{References} & \mathcal{R} \ni r ::= \texttt{x} \mid e\texttt{[}e\texttt{]} \mid e\texttt{[}e\texttt{]}_{\text{js}}
\end{array}
$$

Syntax and Notations. An IR_{ES} program P is a sequence of functions. A function f is defined with its name, parameters, and body instructions with labels. If it is defined with the prefix `syntax`, it is a syntax-directed function, otherwise, a normal function. An instruction i is a reference update, an object allocation, a function call, a branch, or a return instruction. An expression e is a primitive value, a primitive operation, or a reference expression. A reference is a variable, an internal field access, or an external field access. For a given program P, three helper functions func : $\mathcal{L} \to \mathcal{F}$, inst : $\mathcal{L} \to \mathcal{I}$, and next : $\mathcal{L} \to \mathcal{L}$ return the function, instruction, and next label, respectively, of a given label.

$$
\begin{array}{lrl}
\text{States} & \sigma \in \mathbb{S} & = \mathcal{L} \times \mathbb{E} \times \mathbb{C}^* \times \mathbb{H} \\
\text{Environments} & \rho \in \mathbb{E} & = \mathcal{X} \xrightarrow{\text{fin}} \mathbb{V} \\
\text{Calling Contexts} & c \in \mathbb{C} & = \mathcal{L} \times \mathbb{E} \\
\text{Heaps} & h \in \mathbb{H} & = \mathbb{A} \xrightarrow{\text{fin}} \mathcal{L} \times \mathbb{M} \times \mathbb{M}_{\text{js}} \\
\text{Internal Field Maps} & m \in \mathbb{M} & = \mathbb{V}_{\text{str}} \xrightarrow{\text{fin}} \mathbb{V} \\
\text{External Field Maps} & m_{\text{js}} \in \mathbb{M}_{\text{js}} & = \mathbb{V}_{\text{str}} \xrightarrow{\text{fin}} \mathbb{V} \\
\text{Values} & v \in \mathbb{V} & = \mathbb{A} \uplus \mathbb{V}^{\text{p}} \uplus \mathbb{T} \uplus \mathcal{F} \\
\text{Primitive Values} & v^{\text{p}} \in \mathbb{V}^{\text{p}} & = \mathbb{V}_{\text{bool}} \uplus \mathbb{V}_{\text{int}} \uplus \mathbb{V}_{\text{str}} \uplus \cdots \\
\text{JS ASTs} & t \in \mathbb{T} &
\end{array}
$$

Concrete States. An IR_{ES} state $\sigma \in \mathbb{S}$ consists of a label, an environment, a stack of calling contexts, and a heap. An environment $\rho \in \mathbb{E}$ is a finite mapping from variables to values. A calling context $c \in \mathbb{C}$ consists of a label and an environment of the caller. A heap $h \in \mathbb{H}$ is a finite mapping from addresses to labels for allocation sites and two finite mappings from strings to values. The former mapping represents internal fields accessible by $e\texttt{[}e\texttt{]}$, and the latter represents external fields accessible by $e\texttt{[}e\texttt{]}_{\text{js}}$. A value $v \in \mathbb{V}$ is an address, a primitive value (e.g., a boolean b, an integer k, and a string s), a JavaScript AST $t \in \mathbb{T}$, or a function $f \in \mathcal{F}$.

Since IR_{ES} treats JavaScript ASTs as its values, we define them with tree nodes Φ as follows:

$$
\begin{array}{l}
\mathbb{T} \ni t \;::= \tau_k\langle \phi^* \rangle \\
\Phi \ni \phi ::= s \mid t
\end{array}
$$

A JavaScript AST $\tau_k\langle \phi_1, \cdots, \phi_n \rangle$ denotes k-th alternative in the syntactic production of nonterminal symbol τ with multiple tree nodes $\phi_1, \cdots, \phi_n$. A tree node is a string for a terminal symbol or another tree for a nonterminal symbol. We define several notations to easily deal with JavaScript ASTs. The notation τ_k`.eval` denotes an Evaluation function of k-th alternative in the production τ. Similarly, the notation t`.eval` denotes the Evaluation function of the AST t, and it is same with τ_k`.eval` when $t = \tau_k\langle\cdots\rangle$. The Evaluation of each AST takes the AST itself and its tree nodes that are nonterminals as arguments. The notation subs(t) denotes tree nodes that are subtrees of t.

Collecting Semantics. We define denotational semantics of instructions $[\![i]\!] : \mathbb{S} \to \mathbb{S}$ and expressions $[\![e]\!] : \mathbb{S} \to \mathbb{V}$ in Section 1.1 and Section 1.2, respectively. Then, the *collecting semantics* $[\![P]\!]$ of an IR_{ES} program P is a set of reachable states $\mathcal{P}(\mathbb{S})$ from the initial states $\mathbb{S}^\iota \subseteq \mathbb{S}$. We can compute it using a fixpoint algorithm:

$$[\![P]\!] = \lim_{n \to \infty} F^n(\mathbb{S}^\iota)$$

with a *transfer function* $F : \mathcal{P}(\mathbb{S}) \to \mathcal{P}(\mathbb{S})$:

$$F(S) = S \cup \{\sigma' \in \mathbb{S} \mid \sigma \in S \wedge \sigma \rightsquigarrow_P \sigma'\}$$

where $\sigma \rightsquigarrow_P \sigma'$ denotes the one-step transition of a state σ to another state σ' in the program P:

$$\sigma \rightsquigarrow_P \sigma' \iff \sigma = (\ell, \_, \_, \_) \wedge [\![\text{inst}(\ell)]\!](\sigma) = \sigma'$$

Restricted Semantics. Moreover, the *restricted semantics* $[\![P]\!]^{\mathbf{R}} : \mathcal{P}(\mathbb{S}) \to \mathcal{P}(\mathbb{S})$ is a set of reachable states from the initial states restricted by a given set of states:

$$[\![P]\!]^{\mathbf{R}}(S) = \lim_{n\to\infty} F^n(\mathbb{S}^\iota \cap S)$$

1.1 Instructions

$$\boxed{[\![i]\!] : \mathbb{S} \to \mathbb{S}}$$

- Variable Assignments:

$$[\![\mathsf{x} := e]\!](\sigma) = (\mathsf{next}(\ell), \rho[\mathsf{x} \mapsto v], \bar{c}, h)$$

where $\sigma = (\ell, \rho, \bar{c}, h)$ and $[\![e]\!] = v$

- Internal Field Assignments:

$$[\![e_0\texttt{[}e_1\texttt{]} := e_2]\!](\sigma) = (\mathsf{next}(\ell), \rho, \bar{c}, h[a \mapsto (\ell', m', m_{\mathsf{js}})])$$

where

$$\begin{array}{ll} \sigma = (\ell, \rho, \bar{c}, h) & \\ (a, s, v) & = ([\![e_0]\!](\sigma), [\![e_1]\!](\sigma), [\![e_2]\!](\sigma)) \\ (\ell', m, m_{\mathsf{js}}) & = h(a) \\ m' & = m[s \mapsto v] \end{array}$$

- External Field Assignments:

$$[\![e_0\texttt{[}e_1\texttt{]}_{\mathsf{js}} := e_2]\!](\sigma) = (\mathsf{next}(\ell), \rho, \bar{c}, h[a \mapsto (\ell', m, m'_{\mathsf{js}})])$$

where

$$\begin{array}{ll} \sigma & = (\ell, \rho, \bar{c}, h) \\ (a, s, v) & = ([\![e_0]\!](\sigma), [\![e_1]\!](\sigma), [\![e_2]\!](\sigma)) \\ (\ell', m, m_{\mathsf{js}}) & = h(a) \\ m'_{\mathsf{js}} & = m_{\mathsf{js}}[s \mapsto v] \end{array}$$

- Field Mapping Allocations:

$$[\![\mathsf{x} := \texttt{\{\}}]\!](\sigma) = (\mathsf{next}(\ell), \rho[\mathsf{x} \mapsto a], \bar{c}, h[a \mapsto (\ell, \epsilon, \epsilon)])$$

where $\sigma = (\ell, \rho, \bar{c}, h)$ and $a \notin \mathrm{Domain}(h)$

- Function Calls:

$$[\![\mathsf{x} := e(e_1 \cdots e_n)]\!](\sigma) = (\ell', \rho', c :: \bar{c}, h)$$

where

$$\begin{array}{ll} \sigma & = (\ell, \rho, \bar{c}, h) \\ [\![e]\!](\sigma) & = f = \cdots (\mathsf{x}_1, \cdots, \mathsf{x}_n)\ \texttt{\{}\ell' : \cdots\texttt{\}} \\ \rho' & = \bot[\mathsf{x}_1 \mapsto [\![e_1]\!](\sigma), \cdots, \mathsf{x}_n \mapsto [\![e_n]\!](\sigma)] \\ c & = (\ell, \rho) \end{array}$$

- Branches:

$$[\![\texttt{if}\ e\ \ell_{\mathsf{t}}\ \ell_{\mathsf{f}}]\!](\sigma) = \begin{cases} (\ell_{\mathsf{t}}, \rho, \bar{c}, h) & \text{if } [\![e]\!](\sigma) = \texttt{\#t} \\ (\ell_{\mathsf{f}}, \rho, \bar{c}, h) & \text{if } [\![e]\!](\sigma) = \texttt{\#f} \end{cases}$$

- Returns:

$$[\![\texttt{return}\ e]\!](\sigma) = (\mathsf{next}(\ell), \rho[\mathsf{x} \mapsto v], \bar{c}, h)$$

where

$$\begin{array}{ll} \sigma & = (\_, \_, (\ell, \rho) :: \bar{c}, h) \\ [\![e]\!](\sigma) & = v \\ \mathsf{inst}(\ell) & = \mathsf{x} := \cdots \end{array}$$

1.2 Expressions

$$\boxed{[\![e]\!] : \mathbb{S} \to \mathbb{V}}$$

- Primitive Values:

$$[\![v^{\mathsf{p}}]\!](\sigma) = v^{\mathsf{p}}$$

- Primitive Operations:

$$[\![\mathsf{op}(e_1, \cdots, e_n)]\!](\sigma) = \mathsf{op}(v_1^{\mathsf{p}}, \cdots, v_n^{\mathsf{p}})$$

where $\forall 1 \leq j \leq n.\ [\![e_k]\!](\sigma) = v_j^{\mathsf{p}}$

- Variable Lookups:

$$[\![\mathsf{x}]\!](\sigma) = \rho(\mathsf{x})$$

where $\sigma = (\_, \rho, \_, \_)$

- Internal Field Lookups:

$$[\![e_0\texttt{[}e_1\texttt{]}]\!](\sigma) = v$$

where

$$\begin{array}{ll} \sigma & = (\_, \_, \_, h) \\ v_0 & = [\![e_0]\!](\sigma) \\ v_1 & = [\![e_1]\!](\sigma) \\ v & = \begin{cases} m(s) & \text{if } (v_0, v_1) = (a, s) \land h(a) = (\_, m, \_) \\ t_j & \text{if } (v_0, v_1) = (\tau_k\langle t_1, \cdots, t_n\rangle, j) \\ \tau_k.\texttt{eval} & \text{if } (v_0, v_1) = (\tau_k\langle t_1, \cdots, t_n\rangle, \texttt{"eval"}) \end{cases} \end{array}$$

- External Field Lookups:

$$[\![e_0\texttt{[}e_1\texttt{]}_{\mathsf{js}}]\!](\sigma) = v$$

where

$$\begin{array}{ll} \sigma & = (\_, \_, \_, h) \\ (a, s) & = ([\![e_0]\!](\sigma), [\![e_1]\!](\sigma)) \\ h(a) & = (\_, \_, m_{\mathsf{js}}) \\ v & = m_{\mathsf{js}}(s) \end{array}$$

2 JavaScript Definitional Interpreter

In a similar way to $[\![P]\!]$, the collecting semantics $[\![P_{\mathsf{js}}]\!]_{\mathsf{js}}$ of a JavaScript program P_{js} is a set of all reachable JavaScript states $\mathcal{P}(\mathbb{S}_{\mathsf{js}})$ from the initial JavaScript states $\mathbb{S}^\iota_{\mathsf{js}} \subseteq \mathbb{S}_{\mathsf{js}}$. Then, we define a *definitional interpreter* for JavaScript as an $\mathrm{IR}_{\mathrm{ES}}$ program:

Definition 2.1 (JavaScript Definitional Interpreter)**.** An $\mathrm{IR}_{\mathrm{ES}}$ program P is a JavaScript *definitional interpreter* if and only if the following condition holds for each JavaScript program $P_{\mathsf{js}} \in \mathfrak{P}_{\mathsf{js}}$:

$$[\![P_{\mathsf{js}}]\!]_{\mathsf{js}} = \mathsf{decode} \circ [\![P]\!]^{\mathbf{R}} \circ \mathsf{encode}(P_{\mathsf{js}})$$

where $\mathsf{encode} : \mathfrak{P}_{\mathsf{js}} \to \mathcal{P}(\mathbb{S})$ encodes a JavaScript program to $\mathrm{IR}_{\mathrm{ES}}$ states and $\mathsf{decode} : \mathcal{P}(\mathbb{S}) \to \mathcal{P}(\mathbb{S}_{\mathsf{js}})$ decodes $\mathrm{IR}_{\mathrm{ES}}$ states to JavaScript states.

Thus, a restricted semantics of the definitional interpreter with a JavaScript program P_{js} indirectly represents the collecting semantics $[\![P_{\mathsf{js}}]\!]_{\mathsf{js}}$ of the JavaScript program P_{js}. We utilize JISET to automatically extract such JavaScript definitional interpreters from ECMA-262, the standard specification of ECMAScript (the official name of JavaScript) written in English.

3 JavaScript Meta-level Static Analysis

For a JavaScript *meta-level static analysis*, we define an abstract semantics of IR_{ES} in the abstract interpretation framework with *view-based analysis sensitivities* [5, 7].

Abstract Domains. We first define the abstract domain for each structure. We define an analysis sensitivity as a *view abstraction* $\delta : \Pi \to \mathcal{P}(\mathbb{S})$, a function from finite *views* to sets of states. Thus, a sensitive abstract state is defined as a function from pairs of labels and views to abstract states:

- Sensitive Abstract States: $\widehat{\mathbb{D}}_\delta = \mathcal{L} \times \Pi \to \widehat{\mathbb{S}}$

$$
\begin{array}{lcl}
\gamma & : & \widehat{\mathbb{D}}_\delta \to \mathcal{P}(\mathbb{S}) \\
\gamma(\widehat{d_\delta}) & = & \{\sigma \in \mathbb{S} \mid \forall n \geq 0.\ \mathsf{caller}^n(\sigma) = \sigma' \Rightarrow (\\
& & \quad \forall(\ell, \pi) \in \mathcal{L} \times \Pi. \\
& & \quad \sigma' = (\ell, \_, \_, \_) \in \delta(\pi) \Rightarrow \sigma' \in \gamma \circ \widehat{d_\delta}(\pi) \\
& &)\} \\
\widehat{d_\delta} \sqsubseteq \widehat{d'_\delta} & \Leftrightarrow & \forall(\ell, \pi) \in \Pi.\ \widehat{d_\delta}(\ell, \pi) \sqsubseteq \widehat{d'_\delta}(\ell, \pi) \\
\widehat{d_\delta} \sqcup \widehat{d'_\delta} & = & \lambda(\ell, \pi) \in \Pi.\ \widehat{d_\delta}(\ell, \pi) \sqcup \widehat{d'_\delta}(\ell, \pi) \\
\widehat{d_\delta} \sqcap \widehat{d'_\delta} & = & \lambda(\ell, \pi) \in \Pi.\ \widehat{d_\delta}(\ell, \pi) \sqcap \widehat{d'_\delta}(\ell, \pi)
\end{array}
$$

- Abstract States: $\widehat{\mathbb{S}} = \widehat{\mathbb{E}} \times \widehat{\mathbb{C}} \times \widehat{\mathbb{H}}$

$$
\begin{array}{lcl}
\gamma & : & \widehat{\mathbb{S}} \to \mathcal{P}(\mathbb{S}) \\
\gamma(\widehat{\sigma}) & = & \{\sigma \in \mathbb{S} \mid \rho \in \gamma(\widehat{\rho}) \land \sigma \in \gamma(\widehat{c}) \land (h, \_) \in \gamma(\widehat{h})\} \\
& & \quad \text{where } \widehat{\sigma} = (\widehat{\rho}, \widehat{c}, \widehat{h}) \text{ and } \sigma = (\_, \rho, \_, h) \\
\widehat{\sigma} \sqsubseteq \widehat{\sigma}' & \Leftrightarrow & \widehat{\rho} \sqsubseteq \widehat{\rho}' \land \widehat{c} \sqsubseteq \widehat{c}' \land \widehat{h} \sqsubseteq \widehat{h}' \\
\widehat{\sigma} \sqcup \widehat{\sigma}' & = & (\widehat{\rho} \sqcup \widehat{\rho}', \widehat{c} \sqcup \widehat{c}', \widehat{h} \sqcup \widehat{h}' \\
\widehat{\sigma} \sqcap \widehat{\sigma}' & = & (\widehat{\rho} \sqcap \widehat{\rho}', \widehat{c} \sqcap \widehat{c}', \widehat{h} \sqcap \widehat{h}' \\
)
\end{array}
$$

- Abstract Environments: $\widehat{\mathbb{E}} = \mathcal{X} \to \widehat{\mathbb{V}}$

$$
\begin{array}{lcl}
\gamma & : & \widehat{\mathbb{E}} \to \mathcal{P}(\mathbb{E}) \\
\gamma(\widehat{\rho}) & = & \{\rho \in \mathbb{E} \mid \forall \mathsf{x} \mapsto v \in \rho.\ v \in \gamma \circ \widehat{\rho}(\mathsf{x})\} \\
\widehat{\rho} \sqsubseteq \widehat{\rho}' & \Leftrightarrow & \forall \mathsf{x} \in \mathcal{X}.\ \widehat{\rho}(\mathsf{x}) \sqsubseteq \widehat{\rho}'(\mathsf{x}) \\
\widehat{\rho} \sqcup \widehat{\rho}' & \Leftrightarrow & \lambda \mathsf{x} \in \mathcal{X}.\ \widehat{\rho}(\mathsf{x}) \sqcup \widehat{\rho}'(\mathsf{x}) \\
\widehat{\rho} \sqcap \widehat{\rho}' & \Leftrightarrow & \lambda \mathsf{x} \in \mathcal{X}.\ \widehat{\rho}(\mathsf{x}) \sqcap \widehat{\rho}'(\mathsf{x})
\end{array}
$$

- Abstract Contexts: $\widehat{\mathbb{C}} = \mathcal{P}(\mathcal{L} \times \Pi)$

$$
\begin{array}{lcl}
\gamma & : & \widehat{\mathbb{C}} \to \mathcal{P}(\mathbb{S}) \\
\gamma(\widehat{c}) & = & \{\sigma \in \mathbb{S} \mid \mathsf{caller}(\sigma) = \sigma' = (\ell, \_, \_, \_) \Rightarrow \\
& & \quad \exists(\ell, \pi) \in \widehat{c}.\ \sigma' \in \delta(\pi)\} \\
\widehat{c} \sqsubseteq \widehat{c}' & \Leftrightarrow & \widehat{c} \subseteq \widehat{c}' \\
\widehat{c} \sqcup \widehat{c}' & = & \widehat{c} \cup \widehat{c}' \\
\widehat{c} \sqcap \widehat{c}' & = & \widehat{c} \cap \widehat{c}'
\end{array}
$$

- Abstract Heaps: $\widehat{\mathbb{H}} = \widehat{\mathbb{A}} \to \widehat{\mathbb{M}} \times \widehat{\mathbb{M}_{\mathsf{js}}}$

$$
\begin{array}{lcl}
\gamma & : & \widehat{\mathbb{H}} \to \mathcal{P}(\mathbb{H}) \\
\gamma(\widehat{h}) & = & \{h \in \mathbb{H} \mid \forall a \mapsto (\ell, m, m_{\mathsf{js}}) \in h.\ \ell = \eta(a) \land \\
& & \quad (\widehat{m}, \widehat{m_{\mathsf{js}}}) = \widehat{h}(\ell) \land m \in \gamma(\widehat{m}) \land m_{\mathsf{js}} \in \gamma(\widehat{m_{\mathsf{js}}})\} \\
\widehat{h} \sqsubseteq \widehat{h}' & \Leftrightarrow & \forall \widehat{a} \in \widehat{\mathbb{A}}.\ \widehat{m} \sqsubseteq \widehat{m}' \land \widehat{m_{\mathsf{js}}} \sqsubseteq \widehat{m_{\mathsf{js}}}' \\
\widehat{h} \sqcup \widehat{h}' & = & \lambda \widehat{a} \in \widehat{\mathbb{A}}.\ (\widehat{m} \sqcup \widehat{m}', \widehat{m_{\mathsf{js}}} \sqcup \widehat{m_{\mathsf{js}}}') \\
\widehat{h} \sqcap \widehat{h}' & = & \lambda \widehat{a} \in \widehat{\mathbb{A}}.\ (\widehat{m} \sqcap \widehat{m}', \widehat{m_{\mathsf{js}}} \sqcap \widehat{m_{\mathsf{js}}}') \\
& & \quad \text{where } \widehat{h}(\widehat{a}) = (\widehat{m}, \widehat{m_{\mathsf{js}}}) \text{ and } \widehat{h}'(\widehat{a}) = (\widehat{m}', \widehat{m_{\mathsf{js}}}')
\end{array}
$$

- Abstract Internal Field Maps: $\widehat{\mathbb{M}} = \mathbb{V}_{\mathsf{str}} \to \widehat{\mathbb{V}}$

$$
\begin{array}{lcl}
\gamma & : & \widehat{\mathbb{M}} \to \mathcal{P}(\mathbb{M}) \\
\gamma(\widehat{m}) & = & \{m \in \mathbb{M} \mid \forall s \mapsto v \in m. \\
& & \quad v \in \gamma \circ \widehat{m}(s)\} \\
\widehat{m} \sqsubseteq \widehat{m}' & \Leftrightarrow & \forall s \in \mathbb{V}_{\mathsf{str}}.\ \widehat{m}(s) \sqsubseteq \widehat{m}'(s) \\
\widehat{m} \sqcup \widehat{m}' & = & \lambda s \in \mathbb{V}_{\mathsf{str}}.\ \widehat{m}(s) \sqcup \widehat{m}'(s) \\
\widehat{m} \sqcap \widehat{m}' & = & \lambda s \in \mathbb{V}_{\mathsf{str}}.\ \widehat{m}(s) \sqcap \widehat{m}'(s)
\end{array}
$$

- Abstract External Field Maps: $\widehat{\mathbb{M}_{\mathsf{js}}} = \mathbb{V}_{\mathsf{str}} \to \widehat{\mathbb{V}}$

$$
\begin{array}{lcl}
\gamma & : & \widehat{\mathbb{M}_{\mathsf{js}}} \to \mathcal{P}(\mathbb{M}_{\mathsf{js}}) \\
\gamma(\widehat{m_{\mathsf{js}}}) & = & \{m_{\mathsf{js}} \in \mathbb{M}_{\mathsf{js}} \mid \forall s \mapsto v \in m_{\mathsf{js}}. \\
& & \quad v \in \gamma \circ \widehat{m_{\mathsf{js}}}(s)\} \\
\widehat{m_{\mathsf{js}}} \sqsubseteq \widehat{m_{\mathsf{js}}}' & \Leftrightarrow & \forall s \in \mathbb{V}_{\mathsf{str}}.\ \widehat{m_{\mathsf{js}}}(s) \sqsubseteq \widehat{m_{\mathsf{js}}}'(s) \\
\widehat{m_{\mathsf{js}}} \sqcup \widehat{m_{\mathsf{js}}}' & = & \lambda s \in \mathbb{V}_{\mathsf{str}}.\ \widehat{m_{\mathsf{js}}}(s) \sqcup \widehat{m_{\mathsf{js}}}'(s) \\
\widehat{m_{\mathsf{js}}} \sqcap \widehat{m_{\mathsf{js}}}' & = & \lambda s \in \mathbb{V}_{\mathsf{str}}.\ \widehat{m_{\mathsf{js}}}(s) \sqcap \widehat{m_{\mathsf{js}}}'(s)
\end{array}
$$

- Abstract Values: $\widehat{\mathbb{V}} = \mathcal{P}(\widehat{\mathbb{A}} \uplus \mathbb{V}^{\mathsf{p}} \uplus \mathbb{T} \uplus \mathcal{F})$

$$
\begin{array}{lcl}
\gamma & : & \widehat{\mathbb{V}} \to \mathcal{P}(\mathbb{V}) \\
\gamma(\widehat{v}) & = & (\widehat{v} \setminus \widehat{\mathbb{A}}) \uplus \{a \in \mathbb{A} \mid \eta(a) \in \widehat{v}\} \\
\widehat{v} \sqsubseteq \widehat{v}' & \Leftrightarrow & \widehat{v} \subseteq \widehat{v}' \\
\widehat{v} \sqcup \widehat{v}' & = & \widehat{v} \cup \widehat{v}' \\
\widehat{v} \sqcap \widehat{v}' & = & \widehat{v} \cap \widehat{v}'
\end{array}
$$

An abstract state $\widehat{\sigma} \in \widehat{\mathbb{S}}$ consists of an abstract environment, an abstract context, and an abstract heap. An abstract environment $\widehat{\rho} \in \widehat{\mathbb{E}}$ maps variables to abstract values. An abstract context $\widehat{c} \in \widehat{\mathbb{C}}$ is a set of pairs of labels and views for callers. An abstract heap $\widehat{h} \in \widehat{\mathbb{H}}$ is a function from abstract addresses to pairs of abstract internal and external field maps. An abstract field map is a function from strings to abstract values. An abstract address $\widehat{a} \in \widehat{\mathbb{A}}$ is defined with the *allocation-site abstraction* [1], which partitions concrete addresses $\mathbb{A}$ based on their allocation sites $\mathcal{L}$. An abstract value $\widehat{v} \in \widehat{\mathbb{V}}$ is a set of abstract addresses and non-address values. While we use concrete strings in abstract field maps and sets of primitive values in abstract values in this formalization for brevity, we abstract them to bound the height of their lattices as finite in the implementation. We define a partial order $\sqsubseteq$, a join operator $\sqcup$, and a meet operator $\sqcap$. Then, we define the concretization function γ for each abstract domain with the following a helper function $\mathsf{caller} : \mathbb{S} \rightarrowtail \mathbb{S}$

to get callers' states:

$$\sigma = (\_, \_, (\ell, \rho) :: \bar{c}, h) \Rightarrow \mathsf{caller}(\sigma) = (\ell, \rho, \bar{c}, h)$$

and a *valuation* [4] $\eta : \mathbb{A} \to \widehat{\mathbb{A}}$ to correctly concretize abstract addresses.

Abstract Semantics. Using abstract domains, we define the *abstract semantics* $\widehat{[\![P]\!]}$ of an $\mathrm{IR_{ES}}$ program P:

$$\widehat{[\![P]\!]} = \lim_{n\to\infty} \widehat{F}^n(\widehat{d}^\iota_\delta)$$

with an *initial sensitive abstract state* $\widehat{d}^\iota_\delta$ (i.e., $\mathbb{S}^\iota \subseteq \gamma(\widehat{d}^\iota_\delta)$) and an *abstract transfer function* $\widehat{F} : \widehat{\mathbb{D}}_\delta \to \widehat{\mathbb{D}}_\delta$:

$$\widehat{F}(\widehat{d}_\delta) = \widehat{d}_\delta \sqcup \bigsqcup_{(\ell,\pi)\in\mathcal{L}\times\Pi} \delta\widehat{[\![\mathsf{inst}(\ell)]\!]}(\ell, \pi, \widehat{d}_\delta(\ell, \pi))$$

where $\delta\widehat{[\![i]\!]} : \mathcal{L} \times \Pi \times \widehat{\mathbb{S}} \to \widehat{\mathbb{D}}_\delta$ is an abstract semantics of a view abstraction $\delta : \Pi \to \mathcal{P}(\mathbb{S})$.

Restricted Abstract Semantics. Then, we also define the *restricted abstract semantics* $\widehat{[\![P]\!]^{\mathbf{R}}} : \widehat{\mathbb{D}}_\delta \to \widehat{\mathbb{D}}_\delta$ of an $\mathrm{IR_{ES}}$ program P with a given sensitive abstract state:

$$\widehat{[\![P]\!]^{\mathbf{R}}}(\widehat{d}_\delta) = \lim_{n\to\infty} \widehat{F}^n(\widehat{d}^\iota_\delta \sqcap \widehat{d}_\delta)$$

Meta-level Static Analysis. Finally, we define a JavaScript meta-level static analysis using the restricted abstract semantics $\widehat{[\![P]\!]^{\mathbf{R}}}$ of a JavaScript definitional interpreter P:

Definition 3.1 (JavaScript Meta-level Static Analysis)**.** A JavaScript *meta-level static analysis* is a way to indirectly analyze a JavaScript program P_{js} using a restricted abstract semantics $\widehat{[\![P]\!]^{\mathbf{R}}}$ of a JavaScript definitional interpreter P:

$$[\![P_{\mathsf{js}}]\!]_{\mathsf{js}} \subseteq \widehat{\mathsf{decode}} \circ \widehat{[\![P]\!]^{\mathbf{R}}} \circ \widehat{\mathsf{encode}}(P_{\mathsf{js}})$$

where $\widehat{\mathsf{encode}} : \mathfrak{P}_{\mathsf{js}} \to \widehat{\mathbb{D}}_\delta$ encodes a JavaScript program to a sensitive abstract state and $\widehat{\mathsf{decode}} : \widehat{\mathbb{D}}_\delta \to \mathcal{P}(\mathbb{S}_{\mathsf{js}})$ decodes a sensitive abstract state to JavaScript states.

3.1 Flow-Sensitivity for $\mathrm{IR_{ES}}$

We define the flow-sensitivity for $\mathrm{IR_{ES}}$ with a view abstraction $\delta^{\mathsf{flow}} : \{\pi\} \to \mathcal{P}(\mathbb{S})$:

$$\delta^{\mathsf{flow}}(\pi) = \mathbb{S}$$

We define the abstract semantics of the flow-sensitivity for $\mathrm{IR_{ES}}$ as follows:

$$\boxed{\delta^{\mathsf{flow}}\widehat{[\![i]\!]} : \mathcal{L} \times \{\pi\} \times \widehat{\mathbb{S}} \to \widehat{\mathbb{D}}_{\delta^{\mathsf{flow}}}}$$

- <u>Variable Assignments</u>:

$$\delta^{\mathsf{flow}}\widehat{[\![\mathsf{x} := e]\!]}(\ell, \pi, \widehat{\sigma}) = \bot[(\ell', \pi) \mapsto \widehat{\sigma}']$$

where

$$\begin{aligned} \ell' &= \mathsf{next}(\ell) \\ \widehat{\sigma} &= (\widehat{\rho}, \widehat{c}, \widehat{h}) \\ \widehat{v} &= \widehat{[\![e]\!]}(\widehat{\sigma}) \\ \widehat{\sigma}' &= (\widehat{\rho}[\mathsf{x} \mapsto \widehat{v}], \widehat{c}, \widehat{h}) \end{aligned}$$

- <u>Internal Field Assignments</u>:

$$\delta^{\mathsf{flow}}\widehat{[\![e_0[\![e_1]\!] := e_2]\!]}(\ell, \pi, \widehat{\sigma}) = \bot[(\ell', \pi) \mapsto \widehat{\sigma}']$$

where

$$\begin{aligned} \ell' &= \mathsf{next}(\ell) \\ \widehat{\sigma} &= (\widehat{\rho}, \widehat{c}, \widehat{h}) \\ (\widehat{v}_0, \widehat{v}_1, \widehat{v}_2) &= (\widehat{[\![e_0]\!]}(\widehat{\sigma}), \widehat{[\![e_1]\!]}(\widehat{\sigma}), \widehat{[\![e_2]\!]}(\widehat{\sigma})) \\ \widehat{v}_0 \cap \widehat{\mathbb{A}} &= \{\widehat{a}_1, \cdots, \widehat{a}_n\} \\ \widehat{h}' &= \widehat{h}[\widehat{a}_1 \mapsto (\widehat{m}'_1, \widehat{m_{\mathsf{js}}}_1), \cdots, \widehat{a}_n \mapsto (\widehat{m}'_n, \widehat{m_{\mathsf{js}}}_n)] \\ \widehat{v}_1 \cap \mathbb{V}_{\mathsf{str}} &= \{s_1, \cdots, s_m\} \\ \forall 1 \le j \le n&. \\ (\widehat{m}_j, \widehat{m_{\mathsf{js}}}_j) &= \widehat{h}(\widehat{a}_j) \\ \widehat{m}'_j &= \widehat{m}_j \sqcup \bot[s_1 \mapsto \widehat{v}_2, \cdots, s_m \mapsto \widehat{v}_2] \\ \widehat{\sigma}' &= (\widehat{\rho}, \widehat{c}, \widehat{h}') \end{aligned}$$

- <u>External Field Assignments</u>:

$$\delta^{\mathsf{flow}}\widehat{[\![e_0[\![e_1]\!] := e_2]\!]}(\ell, \pi, \widehat{\sigma}) = \bot[(\ell', \pi) \mapsto \widehat{\sigma}']$$

where

$$\begin{aligned} \ell' &= \mathsf{next}(\ell) \\ \widehat{\sigma} &= (\widehat{\rho}, \widehat{c}, \widehat{h}) \\ (\widehat{v}_0, \widehat{v}_1, \widehat{v}_2) &= (\widehat{[\![e_0]\!]}(\widehat{\sigma}), \widehat{[\![e_1]\!]}(\widehat{\sigma}), \widehat{[\![e_2]\!]}(\widehat{\sigma})) \\ \widehat{v}_0 \cap \widehat{\mathbb{A}} &= \{\widehat{a}_1, \cdots, \widehat{a}_n\} \\ \widehat{h}' &= \widehat{h}[\widehat{a}_1 \mapsto (\widehat{m}_1, \widehat{m_{\mathsf{js}}}'_1), \cdots, \widehat{a}_n \mapsto (\widehat{m}_n, \widehat{m_{\mathsf{js}}}'_n)] \\ \widehat{v}_1 \cap \mathbb{V}_{\mathsf{str}} &= \{s_1, \cdots, s_m\} \\ \forall 1 \le j \le n&. \\ (\widehat{m}_j, \widehat{m_{\mathsf{js}}}_j) &= \widehat{h}(\widehat{a}_j) \\ \widehat{m_{\mathsf{js}}}'_j &= \widehat{m_{\mathsf{js}}}_j \sqcup \bot[s_1 \mapsto \widehat{v}_2, \cdots, s_m \mapsto \widehat{v}_2] \\ \widehat{\sigma}' &= (\widehat{\rho}, \widehat{c}, \widehat{h}') \end{aligned}$$

- <u>Object Allocations</u>:

$$\delta^{\mathsf{flow}}\widehat{[\![\mathsf{x} := \{\}]\!]}(\ell, \pi, \widehat{\sigma}) = \bot[(\ell', \pi) \mapsto \widehat{\sigma}']$$

where

$$\begin{aligned} \ell' &= \mathsf{next}(\ell) \\ \widehat{\sigma} &= (\widehat{\rho}, \widehat{c}, \widehat{h}) \\ \widehat{a} &= \ell \\ \widehat{\rho}' &= \widehat{\rho}[\mathsf{x} \mapsto \widehat{a}] \\ \widehat{h}' &= \widehat{h}[\widehat{a} \mapsto (\bot, \bot)] \\ \widehat{\sigma}' &= (\widehat{\rho}', \widehat{c}, \widehat{h}') \end{aligned}$$

- <u>Function Calls</u>:

$$\delta^{\mathsf{flow}}\widehat{[\![\mathsf{x} := e(e_1 \cdots e_n)]\!]}(\ell, \pi, \widehat{\sigma}) = \widehat{d}'_{\delta^{\mathsf{flow}}}$$

where

$$
\begin{array}{ll}
\widehat{\sigma} & = (\widehat{\rho}, \widehat{c}, \widehat{h}) \\
\widehat{v} & = \widehat{[\![e]\!]}(\widehat{\sigma}) \\
\widehat{v}_j & = \widehat{[\![e_j]\!]}(\widehat{\sigma}) \ [\forall 1 \le j \le n] \\
F & = \widehat{v} \cap \mathcal{F} \\
\widehat{d}_{\delta^{\text{flow}}} & = \lambda(\ell', \cdot) \in \mathcal{L} \times \{\pi\}. \\
& \quad \begin{cases} \widehat{\sigma}' & \text{if } \exists f \in F.\ f = \cdots (\mathsf{x}_1, \cdots, \mathsf{x}_n)\ \{\ell' : \cdots\} \\ \bot & \text{otherwise} \end{cases} \\
\widehat{\rho}' & = \bot[\mathsf{x}_1 \mapsto \widehat{v}_1, \cdots, \mathsf{x}_n \mapsto \widehat{v}_n] \\
\widehat{\sigma}' & = (\widehat{\rho}', \{(\ell, \pi)\}, \widehat{h}) \\
\ell'' & = \mathsf{next}(\ell) \\
\widehat{\sigma}'' & = (\widehat{\rho}, \bot, \bot) \\
\widehat{d}'_{\delta^{\text{flow}}} & = \widehat{d}_{\delta^{\text{flow}}}[(\ell'', \cdot) \mapsto \widehat{\sigma}'']
\end{array}
$$

- Branches:

$$\delta^{\text{flow}}\widehat{[\![\mathtt{if}\ e\ \ell_t\ \ell_f]\!]}(\ell, \pi, \widehat{\sigma}) = \widehat{d}'_{\delta^{\text{flow}}}$$

where

$$
\begin{array}{ll}
\widehat{\sigma} & = (\widehat{\rho}, \widehat{c}, \widehat{h}) \\
\widehat{v} & = \widehat{[\![e]\!]}(\widehat{\sigma}) \\
\widehat{d}_{\delta^{\text{flow}}} & = \begin{cases} \bot[\ell_t \mapsto \widehat{\sigma}] & \text{if } \#\mathsf{t} \in \widehat{v} \\ \bot & \text{otherwise} \end{cases} \\
\widehat{d}'_{\delta^{\text{flow}}} & = \begin{cases} \widehat{d}_{\delta}[\ell_f \mapsto \widehat{\sigma}] & \text{if } \#\mathsf{f} \in \widehat{v} \\ \widehat{d}_{\delta} & \text{otherwise} \end{cases}
\end{array}
$$

- Returns:

$$\delta^{\text{flow}}\widehat{[\![\mathtt{return}\ e]\!]}(\ell, \pi, \widehat{\sigma}) = \widehat{d}_{\delta^{\text{flow}}}$$

where

$$
\begin{array}{ll}
\widehat{\sigma} & = (\widehat{\rho}, \widehat{c}, \widehat{h}) \\
\widehat{v} & = \widehat{[\![e]\!]}(\widehat{\sigma}) \\
\widehat{d}_{\delta^{\text{flow}}} & = \lambda(\ell', \cdot) \in \mathcal{L} \times \{\pi\}. \\
& \quad \begin{cases} \bot[\mathsf{x} \mapsto \widehat{v}] & \text{if } \exists(\ell', \cdot) \in \widehat{c} \wedge \mathsf{inst}(\ell') = \mathsf{x} := \cdots \\ \bot & \text{otherwise} \end{cases}
\end{array}
$$

where $\widehat{[\![e]\!]} : \widehat{\mathbb{S}} \to \widehat{\mathbb{V}}$ is an abstract semantics of expressions:

- Primitive Values:

$$\widehat{[\![v^{\mathsf{p}}]\!]}(\widehat{\sigma}) = \{v^{\mathsf{p}}\}$$

- Primitive Operations:

$$\widehat{[\![\mathsf{op}(e_1, \cdots, e_n)]\!]}(\widehat{\sigma}) = \widehat{v}$$

where

$$
\begin{array}{ll}
\widehat{[\![e_j]\!]}(\widehat{\sigma}) & = \widehat{v}_j \ [\forall 1 \le j \le n] \\
\widehat{v} & = \widehat{\mathsf{op}}(\widehat{v}_1 \cap \mathbb{V}^{\mathsf{p}}, \cdots, \widehat{v}_n \cap \mathbb{V}^{\mathsf{p}})
\end{array}
$$

- Variable Lookups:

$$\widehat{[\![\mathsf{x}]\!]}(\widehat{\sigma}) = \widehat{\rho}(\mathsf{x})$$

where $\widehat{\sigma} = (\widehat{\rho}, \_, \_)$

- Internal Field Lookups:

$$\widehat{[\![e_0[e_1]]\!]}(\widehat{\rho}) = \widehat{v}$$

where

$$
\begin{array}{lll}
\widehat{[\![e_0]\!]}(\widehat{\rho}) & = & \widehat{v}_0 \\
\widehat{[\![e_1]\!]}(\widehat{\rho}) & = & \widehat{v}_1 \\
\widehat{v} & = & \text{(a point-wise internal field lookup definition} \\
& & \text{with } \widehat{v}_0 \text{ and } \widehat{v}_1\text{)}
\end{array}
$$

- External Field Lookups:

$$\widehat{[\![e_0[e_1]]\!]}(\widehat{\rho}) = \widehat{v}$$

where

$$
\begin{array}{lll}
\widehat{[\![e_0]\!]}(\widehat{\rho}) & = & \widehat{v}_0 \\
\widehat{[\![e_1]\!]}(\widehat{\rho}) & = & \widehat{v}_1 \\
\widehat{v} & = & \text{(a point-wise external field lookup definition} \\
& & \text{with } \widehat{v}_0 \text{ and } \widehat{v}_1\text{)}
\end{array}
$$

3.2 Callsite-Sensitivity for IR_{ES}

We define the *callsite-sensitivity* [8, 9] for IR_{ES} with a view abstraction $\delta^{k\text{-cfa}} : \mathcal{L}^{\le k} \to \mathcal{P}(\mathbb{S})$:

$$
\begin{array}{l}
\delta^{k\text{-cfa}}([\ell_1, \cdots, \ell_n]) = \{\sigma = (\_, \_, [c_1, \cdots, c_m], \_) \in \mathbb{S} \mid \\
\qquad (n = k \le m \vee n = m) \wedge \forall 1 \le i \le n.\ c_i = (\ell_i, \_)\}
\end{array}
$$

We define the abstract semantics of the callsite-sensitivity for IR_{ES} by modifying that of the flow-sensitivity for IR_{ES} as follows:

$$\boxed{\delta^{k\text{-cfa}}\widehat{[\![i]\!]} : \mathcal{L} \times \mathcal{L}^{\le k} \times \widehat{\mathbb{S}} \to \widehat{\mathbb{D}}_{\delta^{k\text{-cfa}}}}$$

- Function Calls:

$$\delta^{k\text{-cfa}}\widehat{[\![\mathsf{x} := e(e_1 \cdots e_n)]\!]}(\ell, [\ell_1, \cdots, \ell_n], \widehat{\sigma}) = \widehat{d}'_{\delta^{k\text{-cfa}}}$$

where

$$
\begin{array}{l}
\cdots \\
\widehat{d}_{\delta^{k\text{-cfa}}} = \lambda(\ell', [\ell'_1, \cdots, \ell'_m]) \in \mathcal{L} \times \mathcal{L}^{\le k}. \\
\quad \begin{cases} \widehat{\sigma}' & \text{if } \exists f \in F.\ f = \cdots (\mathsf{x}_1, \cdots, \mathsf{x}_n)\ \{\ell' : \cdots\} \\ & \quad \begin{pmatrix} n = k = m \wedge \\ [\ell', \ell_1, \cdots, \ell_n] = [\ell'_1, \cdots, \ell'_m, \ell_n] \end{pmatrix} \vee \\ & \quad \begin{pmatrix} m = n + 1 \wedge \\ [\ell', \ell_1, \cdots, \ell_n] = [\ell'_1, \cdots, \ell'_m] \end{pmatrix} \\ \bot & \text{otherwise} \end{cases} \\
\cdots
\end{array}
$$

4 Analysis Sensitivities for JavaScript

In a JavaScript meta-level static analysis, analysis sensitivities for JavaScript are different from those for IR_{ES}. For example, let us explain the analysis of the following JavaScript code with the flow-sensitivity for IR_{ES}:

```
let x = 1, y = 2;        x + y; // 3
```

13.1.3 Runtime Semantics: Evaluation

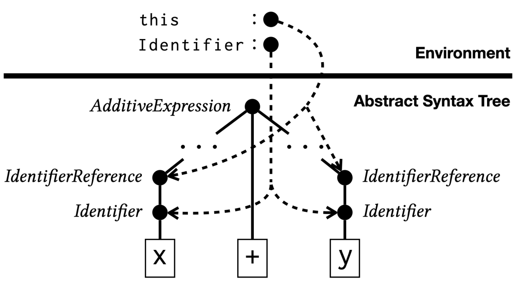
IdentifierReference **:** *Identifier*

1. Return ? ResolveBinding(StringValue of *Identifier*).

(a) Evaluation algorithm for identifier references

```
syntax def
    IdentifierReference[0].Evaluation(
  this, Identifier
) {
  return [? (ResolveBinding
      (Identifier.StringValue))]
}
```

(b) Extracted $\mathrm{IR}_{\mathrm{ES}}$ function for identifier references



(c) Result of `x + y` via a definitional interpreter

Figure 1. A JavaScript meta-level static analysis with the flow-sensitivity for $\mathrm{IR}_{\mathrm{ES}}$

Figure 1 shows (a) the Evaluation algorithm of identifier references, (b) its extracted $\mathrm{IR}_{\mathrm{ES}}$ function, and (c) the parsing result of `x + y` and the initial local environment of the $\mathrm{IR}_{\mathrm{ES}}$ function. Since the flow-sensitivity merges states on the same labels, contexts for the evaluation of both identifier references `x` and `y` are merged. Thus, the $\mathrm{IR}_{\mathrm{ES}}$ variable `Identifier` points to their ASTs as illustrated at the bottom of Figure 1(c). Due to the imprecise merge of contexts, StringValue of `Identifier` returns `"x"` and `"y"`, and ResolveBinding with them returns both `1` and `2`. Finally, the analysis result of `x + y` becomes `{ 2, 3, 4 }`.

4.1 Flow-Sensitivity for JavaScript

To resolve this problem, we present an *AST sensitivity* for $\mathrm{IR}_{\mathrm{ES}}$ as a variant of *object sensitivity* [6, 10] to represent flow-sensitivity for JavaScript. The object sensitivity uses abstract addresses $\widehat{\mathbb{A}}$ of receiver objects as views. However, the AST sensitivity utilizes JavaScript ASTs $\mathbb{T}$ stored in `this` parameter for syntax-directed functions as views with a view abstraction $\delta^{\text{js-flow}} : \mathbb{T} \uplus \{\bot\} \rightarrow \mathcal{P}(\mathbb{S})$:

$$\delta^{\text{js-flow}}(t_\bot) = \{\sigma = (\_, \_, \overline{c}, \_) \in \mathbb{S} \mid \mathsf{ast}(\overline{c}) = t_\bot\}$$

where $\mathsf{ast} : \mathbb{C}^* \rightarrow \mathbb{T} \uplus \{\bot\}$ denotes the JavaScript AST stored in `this` parameter of the top-most syntax-directed function for a given calling context stack:

$$\mathsf{ast}(\overline{c}) = \begin{cases} t \;\; \text{if } \exists c.\ \overline{c} = c_1 :: \cdots :: c_n :: c :: \cdots \wedge c = (\ell, \rho) \wedge \\ \qquad \mathsf{func}(\ell) = \texttt{syntax def} \cdots \wedge \rho(\texttt{this}) = t \wedge \\ \qquad \forall 1 \le j \le n.\ c_j = (\ell_j, \_) \wedge \mathsf{func}(\ell_j) = \texttt{def} \cdots \\ \bot \text{ otherwise} \end{cases}$$

Note that the number of views for the AST sensitivity is finite as well because JavaScript ASTs are finite in a JavaScript program. We define the flow-sensitivity for JavaScript using the AST sensitivity for $\mathrm{IR}_{\mathrm{ES}}$. It successfully divides contexts for the evaluation of JavaScript identifiers `x` and `y` in the example even though their labels in $\mathrm{IR}_{\mathrm{ES}}$ are the same.

We define the abstract semantics of the flow-sensitivity for JavaScript as follows:

$$\boxed{\delta^{\text{js-flow}}\widehat{[\![i]\!]} : \mathcal{L} \times (\mathbb{T} \uplus \{\bot\}) \times \widehat{\mathbb{S}} \rightarrow \widehat{\mathbb{D}}_{\delta^{\text{js-flow}}}}$$

- Function Calls:

$$\delta^{\text{js-flow}}[\![\texttt{x} := \widehat{e(e_1 \cdots e_n)}]\!](\ell, t_\bot, \widehat{\sigma}) = \widehat{d'}_{\delta^{\text{js-flow}}}$$

  where

...

$$\widehat{d}_{\delta^{\text{js-flow}}} = \lambda(\ell', t'_\bot) \in \mathcal{L} \times (\mathbb{T} \uplus \{\bot\}).$$
$$\begin{cases} \widehat{\sigma}' & \text{if } \exists f \in F.\ f = \cdots (\texttt{x}_1, \cdots, \texttt{x}_n)\ \{\ell' : \cdots\} \\ & \qquad \begin{pmatrix} f = \texttt{syntax def} \cdots \wedge \\ t'_\bot \in \widehat{v}_1 \end{pmatrix} \vee \\ & \qquad \begin{pmatrix} f = \texttt{def} \cdots \wedge \\ t_\bot = t'_\bot \end{pmatrix} \\ \bot & \text{otherwise} \end{cases}$$

...

4.2 Callsite-Sensitivity for JavaScript

We also formally define the callsite-sensitivity for JavaScript by extending the AST sensitivity for specific normal $\mathrm{IR}_{\mathrm{ES}}$ functions. In ECMA-262, all explicit and even implicit JavaScript function calls invoke normal $\mathrm{IR}_{\mathrm{ES}}$ functions Call and Construct. Thus, we define the callsite-sensitivity for JavaScript by extending the AST sensitivity with two normal $\mathrm{IR}_{\mathrm{ES}}$ functions with a view abstraction $\delta^{\text{js-}k\text{-cfa}} : \mathbb{T}^{\le k} \rightarrow \mathcal{P}(\mathbb{S})$:

$$\begin{aligned} \delta^{\text{js-}k\text{-cfa}}([t_1, \cdots, t_n]) = \{&\sigma = (\_, \_, \overline{c}, \_) \in \mathbb{S} \mid \\ &n \le k \wedge (n = k \vee \mathsf{js\text{-}ctxt}^{n+1}(\overline{c}) = \bot) \wedge \\ &\forall 1 \le i \le n.\ \mathsf{ast} \circ \mathsf{js\text{-}ctxt}^{i}(\overline{c}) = t_i\} \end{aligned}$$

where $\mathsf{js\text{-}ctxt} : \mathbb{C}^* \rightarrow \mathbb{C}^* \uplus \{\bot\}$ pops out calling contexts until the function of the top-most context is Call or Construct:

$$\mathsf{js\text{-}ctxt}(\overline{c}) = \begin{cases} \overline{c} & \text{if } \overline{c} = (\ell, \rho) :: \_ \wedge \\ & \quad (\mathsf{func}(\ell) = \texttt{def Call} \cdots \vee \\ & \quad \mathsf{func}(\ell) = \texttt{def Construct} \cdots) \\ \mathsf{js\text{-}ctxt}(\overline{c}') & \text{if } \overline{c} = \_ :: \overline{c}' \\ \bot & \text{otherwise} \end{cases}$$

Using this callsite-sensitivity for JavaScript, the meta-level static analyzer can discriminate not only explicit JavaScript function calls (e.g. `f()`) but also implicit JavaScript function calls, including getters/setters, user-defined implicit conversions, and implicit function calls in built-in libraries.

We define the abstract semantics of the callsite-sensitivity for JavaScript as follows:

$$\boxed{\delta^{\text{js-}k\text{-cfa}}\widehat{[\![i]\!]} : \mathcal{L} \times \mathbb{T}^{\leq k} \times \widehat{\mathbb{S}} \to \widehat{\mathbb{D}}_{\delta^{\text{js-}k\text{-cfa}}}}$$

- Function Calls:

$$\delta^{\text{js-}k\text{-cfa}}[\![\mathtt{x} := \widehat{\mathtt{e}(e_1 \cdots e_n)}]\!](\ell, [t_1, \cdots, t_n], \widehat{\sigma}) = \widehat{d}'_{\delta^{\text{js-}k\text{-cfa}}}$$

where

$$\cdots$$

$$\widehat{d}_{\delta^{\text{js-}k\text{-cfa}}} = \lambda(\ell', [t'_1, \cdots, t'_m]) \in \mathcal{L} \times \mathbb{T}^{\leq k}.$$

$$\begin{cases} \widehat{\sigma}' & \text{if } \exists f \in F.\ f = \cdots(\mathtt{x}_1, \cdots, \mathtt{x}_n)\ \{\ell' : \cdots\} \\ & t' = \text{(an AST of the flow-sensitivity for JavaScript)} \\ & \begin{pmatrix} (f = \mathtt{def}\ \text{Call} \cdots \vee \\ \ f = \mathtt{def}\ \text{Construct} \cdots) \wedge \\ n = k = m \wedge \\ [t', t_1, \cdots, t_n] = [t'_1, \cdots, t'_m, t_n] \end{pmatrix} \vee \\ & \begin{pmatrix} (f = \mathtt{def}\ \text{Call} \cdots \vee \\ \ f = \mathtt{def}\ \text{Construct} \cdots) \wedge \\ m = n + 1 \wedge \\ [t', t_1, \cdots, t_n] = [t'_1, \cdots, t'_m] \end{pmatrix} \vee \\ & \begin{pmatrix} \neg(f = \mathtt{def}\ \text{Call} \cdots \vee \\ \quad f = \mathtt{def}\ \text{Construct} \cdots) \wedge \\ m = n \wedge \\ [t_1, \cdots, t_n] = [t'_1, \cdots, t'_m] \end{pmatrix} \\ \bot & \text{otherwise} \end{cases}$$

$$\cdots$$

References

[1] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. 1990. Analysis of Pointers and Structures. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation (PLDI)* (White Plains, New York, USA) *(PLDI '90)*. Association for Computing Machinery, New York, NY, USA, 296–310. https://doi.org/10.1145/93542.93585

[2] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming languages (POPL)*. https://doi.org/10.1145/512950.512973

[3] Patrick Cousot and Radhia Cousot. 1992. Abstract Interpretation Frameworks. *Journal of Logic and Computation (JLC)* 2, 4 (1992), 511–547. https://doi.org/10.1093/logcom/2.4.511

[4] Arlen Cox, Bor-Yuh Evan Chang, and Xavier Rival. 2014. Automatic Analysis of Open Objects in Dynamic Language Programs. In *Proceedings of the 21st International Symposium on Static Analysis (SAS)*. https://doi.org/10.1007/978-3-319-10936-7_9

[5] Se-Won Kim, Xavier Rival, and Sukyoung Ryu. 2018. A Theoretical Foundation of Sensitivity in an Abstract Interpretation Framework. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 40, 3, Article 13 (2018), 44 pages. https://doi.org/10.1145/3230624

[6] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. 2005. Parameterized Object Sensitivity for Points-to Analysis for Java. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 14, 1 (Jan. 2005), 1–41. https://doi.org/10.1145/1044834.1044835

[7] Jihyeok Park, Hongki Lee, and Sukyoung Ryu. 2021. A Survey of Parametric Static Analysis. *ACM Computing Surveys (CSUR)* 54, 7 (2021), 1–37. https://doi.org/10.1145/3464457

[8] Micha Sharir and Amir Pnueli. 1981. *Two Approaches to Interprocedural Data Flow Analysis*.

[9] Olin Grigsby Shivers. 1991. *Control-Flow Analysis of Higher-Order Languages*. Carnegie Mellon University.

[10] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. 2011. Pick Your Contexts Well: Understanding Object-Sensitivity. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)* (Austin, Texas, USA). Association for Computing Machinery, New York, NY, USA, 17–30. https://doi.org/10.1145/1926385.1926390