Type Analysis for a Modified IR$_{\text{ES}}$

Anonymous Author(s)

Abstract—**This technical report is a companion report of the research paper for JSTAR, a JavaScript Specification Type Analyzer using Refinement. In this report, we formally define the syntax and semantics of a modified IR$_{\text{ES}}$, an untyped intermediate representation for ECMAScript. Moreover, we formally define type analysis for the modified IR$_{\text{ES}}$ based on the abstract interpretation framework with flow- and type-sensitivity for arguments. To increase the precision of the type analysis, we also present *condition-based refinement* for type analysis, which prunes out infeasible abstract states using conditions of assertions and branches.**

I. SYNTAX

We first define syntax of the modified IR$_{\text{ES}}$ as follows:

Functions	$\mathbb{F} \ni f$	$::= \mathtt{def}\ \mathtt{x}(\mathtt{x}^*, [\mathtt{x}^*]).\ \ell$
Instructions	$\mathbb{I} \ni i$	$::= \mathtt{let}\ \mathtt{x} = e \mid \mathtt{x} = (e\ e^*) \mid \mathtt{assert}\ e$
		$\mid \mathtt{if}\ e\ \ell\ \ell \mid \mathtt{return}\ e \mid r = e$
References		$r ::= \mathtt{x} \mid r[e]$
Expressions		$e ::= t\ \{[\mathtt{x}:e]^*\} \mid [e^*] \mid e:\tau \mid r?$
		$\mid e \oplus e \mid \ominus e \mid r \mid c \mid p$
Primitives	$\mathbb{P} \ni p$	$::= \mathtt{undefined} \mid \mathtt{null} \mid b \mid n \mid j \mid s \mid \mathtt{@}s$
Types	$\mathbb{T} \ni \tau$	$::= t \mid \mathtt{[]} \mid [\tau] \mid \mathtt{js} \mid \mathtt{prim}$
		$\mid \mathtt{undefined} \mid \mathtt{null} \mid \mathtt{bool} \mid \mathtt{numeric}$
		$\mid \mathtt{num} \mid \mathtt{bigint} \mid \mathtt{str} \mid \mathtt{symbol}$

A modified IR$_{\text{ES}}$ program $P = (\mathtt{func}, \mathtt{inst}, \mathtt{next})$ consists of three mappings; $\mathtt{func} : \mathbb{L} \to \mathbb{F}$ maps labels to their functions, $\mathtt{inst} : \mathbb{L} \to \mathbb{I}$ maps labels to their instructions, and $\mathtt{next} : \mathbb{L} \to \mathbb{L}$ maps labels to their next labels, where a label $\ell \in \mathbb{L}$ denotes a program point. A function $\mathtt{def}\ \mathtt{f}(\mathtt{x}^*, [\mathtt{y}^*]).\ \ell \in \mathbb{F}$ consists of its name $\mathtt{f}$, normal parameters $\mathtt{x}^*$, optional parameters $\mathtt{y}^*$, and a body label ℓ. For presentation brevity, we assume that no global variables exist in this paper. An instruction i is a variable declaration, a function call, an assertion, a branch, a return, or a reference update. An invocation of an abstract algorithm in ECMAScript is compiled to a function call instruction with a new temporary variable. We represent loops using branch instructions with cyclic pointing of labels in $\mathtt{next}$. A reference r is a variable $\mathtt{x}$ or a field access $r[e]$. We write $r.\mathtt{f}$ to briefly represent $r[\mathtt{"f"}]$. An expression e is a record, a list, a type check, an existence check, a binary operation, a unary operation, a reference, a constant, or a primitive, which is either $\mathtt{undefined}$, $\mathtt{null}$, a Boolean b, a Number n, a BigInt j, a String s, or a Symbol $\mathtt{@}s$.

A type $\tau \in \mathbb{T}$ is either a nominal type t, an empty list type $\mathtt{[]}$, a parametric list type $[\tau]$, a JavaScript type $\mathtt{js}$, a primitive type $\mathtt{prim}$, a numeric type $\mathtt{numeric}$, $\mathtt{num}$, $\mathtt{bigint}$, $\mathtt{str}$, or $\mathtt{symbol}$. The subtype relation $<: \subseteq \mathbb{T} \times \mathbb{T}$ between types is reflexive and transitive.

II. SEMANTICS

In this section, we formally define the semantics of the modified IR$_{\text{ES}}$. We will define states $\mathbb{S}$ (Section II-A), and then define a denotational semantics of the modified IR$_{\text{ES}}$ for instructions $[\![i]\!]_i : \mathbb{S} \to \mathbb{S}$ (Section II-B), references $[\![r]\!]_r : \mathbb{S} \to \mathbb{S} \times \mathbb{V}$ (Section II-C), and expressions $[\![e]\!]_e : \mathbb{S} \to \mathbb{S} \times \mathbb{V}$ (Section II-D).

A. States: $\mathbb{S}$

We define states as follows:

States	$d \in$	$\mathbb{S} = \mathbb{L} \times \mathbb{C}^* \times \mathbb{H} \times \mathbb{E}$
Contexts	$\kappa \in$	$\mathbb{C} = \mathbb{L} \times \mathbb{E} \times \mathbb{X}$
Heaps	$h \in$	$\mathbb{H} = \mathbb{A} \to \mathbb{O}$
Addresses	$a \in$	$\mathbb{A}$
Objects	$o \in$	$\mathbb{O} = (\mathbb{T}_t \times (\mathbb{V}_s \to \mathbb{V})) \uplus \mathbb{V}^*$
Nominal Types	$t \in$	$\mathbb{T}_t$
Environments	$\sigma \in$	$\mathbb{E} = \mathbb{X} \times \mathbb{V}$
Values	$v \in$	$\mathbb{V} = \mathbb{F} \uplus \mathbb{A} \uplus \mathbb{V}_c \uplus \mathbb{P}$
Constants	$c \in$	$\mathbb{V}_c$
Strings	$s \in$	$\mathbb{V}_s$

A state $d \in \mathbb{S}$ consists of a label, a context stack, a heap, and an environment. A context $\kappa \in \mathbb{C}$ is a triple of a label, an environment, and a variable. A heap $h \in \mathbb{H}$ is a mapping from addresses to objects. For each address $a \in \mathbb{A}$, an object $o \in \mathbb{O}$ is a record from fields to values with its nominal type or a list of values. An environment $\sigma \in \mathbb{E}$ is a mapping from variables to values. A value $v \in \mathbb{V}$ is a function, an address, a constant, or a primitive value.

B. Instructions: $[\![i]\!]_i : \mathbb{S} \to \mathbb{S}$

- Variable Declarations:

$$[\![\mathtt{let}\ \mathtt{x} = e]\!]_i(d) = (\mathtt{next}(\ell), \overline{\kappa}, h, \sigma[\mathtt{x} \mapsto v])$$

  where

$$[\![e]\!]_e(d) = ((\ell, \overline{\kappa}, h, \sigma), v)$$

- Function Calls:

$$[\![\mathtt{x} = (e_0\ e_1 \cdots e_n)]\!]_i(d) = (\ell_{\mathtt{f}}, \kappa :: \overline{\kappa}, h, \sigma')$$

  where

$$[\![e_0]\!]_e(d) = (d_0, \mathtt{def}\ \mathtt{f}(\mathtt{p}_1, \mathtt{p}_m).\ \ell_{\mathtt{f}}) \wedge$$
$$[\![e_1]\!]_e(d_0) = (d_1, v_1) \wedge \cdots \wedge [\![e_n]\!]_e(d_{n-1}) = (d_n, v_n) \wedge$$
$$d_n = (\ell, \overline{\kappa}, h, \sigma) \wedge k = \min(n, m) \wedge$$
$$\sigma' = [\mathtt{p}_1 \mapsto v_1, \cdots, \mathtt{p}_k \mapsto v_k] \wedge \kappa = (\mathtt{next}(\ell), \sigma, \mathtt{x})$$

- Assertions:

$$[\![\mathtt{assert}\ e]\!]_i(d) = d' \quad \text{if}\ [\![e]\!]_e(d) = (d', \#\mathtt{t})$$

- Branches:

$$\llbracket \texttt{if } e \; \ell_{\texttt{t}} \; \ell_{\texttt{f}} \rrbracket_i(d) = \begin{cases} (\ell_{\texttt{t}}, \overline{\kappa}, h, \sigma) & \text{if } v = \texttt{\#t} \\ (\ell_{\texttt{f}}, \overline{\kappa}, h, \sigma) & \text{if } v = \texttt{\#f} \end{cases}$$

where

$$\llbracket e \rrbracket_e(d) = ((\ell_{\texttt{t}}, \overline{\kappa}, h, \sigma), v)$$

- Returns:

$$\llbracket \texttt{return } e \rrbracket_i(d) = (\ell, \overline{\kappa}, h, \sigma[\texttt{x} \mapsto v])$$

where

$$\llbracket e \rrbracket_e(d) = ((\_, (\ell, \sigma, \texttt{x}) :: \overline{\kappa}, h, \_), v)$$

- Variable Updates:

$$\llbracket \texttt{x} = e \rrbracket_i(d) = (\texttt{next}(\ell), \overline{\kappa}, h, \sigma[\texttt{x} \mapsto v])$$

where

$$\llbracket e \rrbracket_e(d) = ((\ell, \overline{\kappa}, h, \sigma), v)$$

- Field Updates:

$$\llbracket r\,\texttt{[}e_0\texttt{]} = e_1 \rrbracket_i(d) = (\texttt{next}(\ell), \overline{\kappa}, h[a \mapsto o'], \sigma)$$

where

$$\llbracket r \rrbracket_e(d) = (d', a) \wedge \llbracket e_0 \rrbracket_e(d') = (d_0, v_0) \wedge$$
$$\llbracket e_1 \rrbracket_e(d_0) = ((\ell, \overline{\kappa}, h, \sigma), v_1) \wedge o = h(a) \wedge$$
$$o' = \begin{cases} o_r & \text{if } o = (t, \texttt{fs}) \wedge v_0 = s \\ o_l & \text{if } o = [v'_1, \cdots, v'_m] \wedge v_0 = n \end{cases} \wedge$$
$$o_r = (t, \texttt{fs}[s \mapsto v_1]) \wedge o_l = [\cdots, v'_{n-1}, v_1, v'_{n+1}, \cdots]$$

C. References: $\llbracket r \rrbracket_r : \mathbb{S} \to \mathbb{S} \times \mathbb{V}$

- Variable Lookups:

$$\llbracket \texttt{x} \rrbracket_r(d) = (d, \sigma(\texttt{x}))$$

where

$$d = (\_, \_, \_, \sigma)$$

- Field Lookups:

$$\llbracket r\,\texttt{[}e\texttt{]} \rrbracket_r(d) = (d'', v')$$

where

$$\llbracket r \rrbracket_e(d) = (d', a) \wedge \llbracket e \rrbracket_e(d') = (d'', v) \wedge$$
$$d'' = (\ell, \overline{\kappa}, h, \sigma) \wedge o = h(a) \wedge$$
$$v' = \begin{cases} \texttt{fs}(s) & \text{if } o = (t, \texttt{fs}) \wedge v = s \\ v'_n & \text{if } o = [v'_1, \cdots, v'_m] \wedge v = n \\ n & \text{if } o = [v'_1, \cdots, v'_n] \wedge v = \texttt{"length"} \end{cases}$$

D. Expressions: $\llbracket e \rrbracket_e : \mathbb{S} \to \mathbb{S} \times \mathbb{V}$

- Records:

$$\llbracket t \; \texttt{\{}\texttt{x}_1 : e_1, \cdots, \texttt{x}_n : e_n\texttt{\}} \rrbracket_e(d) = (d', a)$$

where

$$\llbracket e_1 \rrbracket_e(d) = (d_1, v_1) \wedge \cdots \wedge \llbracket e_n \rrbracket_e(d_{n-1}) = (d_n, v_n) \wedge$$
$$d_n = (\ell, \overline{\kappa}, h, \sigma) \wedge \texttt{fs} = [\texttt{x}_1 \mapsto v_1, \cdots, \texttt{x}_n \mapsto v_n]$$
$$a \notin \text{Domain}(h) \wedge d' = (\ell, \overline{\kappa}, h[a \mapsto (t, \texttt{fs})], \sigma)$$

- Lists:

$$\llbracket \texttt{[}e_1, \cdots, e_n\texttt{]} \rrbracket_e(d) = (d', a)$$

where

$$\llbracket e_1 \rrbracket_e(d) = (d_1, v_1) \wedge \cdots \wedge \llbracket e_n \rrbracket_e(d_{n-1}) = (d_n, v_n) \wedge$$
$$d_n = (\ell, \overline{\kappa}, h, \sigma) \wedge a \notin \text{Domain}(h) \wedge$$
$$d' = (\ell, \overline{\kappa}, h[a \mapsto [v_1, \cdots, v_n]], \sigma)$$

- Type Checks:

$$\llbracket e : \tau \rrbracket_e(d) = (d', b)$$

where

$$\llbracket e \rrbracket_e(d) = (d', v) \wedge b = \begin{cases} \texttt{\#t} & \text{if } v \text{ is a value of } \tau \\ \texttt{\#f} & \text{otherwise} \end{cases}$$

- Variable Existence Checks:

$$\llbracket \texttt{x?} \rrbracket_e(d) = (d, b)$$

where

$$d = (\_, \_, \_, \sigma) \wedge b = \begin{cases} \texttt{\#t} & \text{if } \texttt{x} \in \text{Domain}(\sigma) \\ \texttt{\#f} & \text{otherwise} \end{cases}$$

- Field Existence Checks:

$$\llbracket r\,\texttt{[}e\texttt{]?} \rrbracket_e(d) = (d'', b)$$

where

$$\llbracket r \rrbracket_e(d) = (d', a) \wedge \llbracket e \rrbracket_e(d') = (d'', v) \wedge$$
$$d'' = (\ell, \overline{\kappa}, h, \sigma) \wedge o = h(a) \wedge$$
$$b = \begin{cases} \texttt{\#t} & \text{if } o = (t, \texttt{fs}) \wedge v = s \wedge s \in \text{Domain}(\texttt{fs}) \\ \texttt{\#t} & \text{if } o = [v'_1, \cdots, v'_m] \wedge v = n \wedge 1 \le n \le m \\ \texttt{\#f} & \text{otherwise} \end{cases}$$

- Binary Operations:

$$\llbracket e \oplus e \rrbracket_e(d) = (d'', v_0 \oplus v_1)$$

where

$$\llbracket e_0 \rrbracket_e(d) = (d', v_0) \wedge \llbracket e_1 \rrbracket_e(d') = (d'', v_1)$$

- Unary Operations:

$$\llbracket \ominus e \rrbracket_e(d) = (d', \ominus v)$$

where

$$\llbracket e \rrbracket_e(d) = (d', v)$$

- References:

$$\llbracket r \rrbracket_e(d) = \llbracket r \rrbracket_r(d)$$

- Constants:

$$\llbracket c \rrbracket_e(d) = (d, c)$$

- Primitives:

$$\llbracket p \rrbracket_e(d) = (d, p)$$

III. TYPE ANALYSIS

We design a type analysis for the modified IR$_{\text{ES}}$ based on the abstract interpretation framework with analysis sensitivity. We will define abstract states $\mathbb{S}^\sharp$ (Section III-A), and then define an abstract semantics of the modified IR$_{\text{ES}}$ for instructions $\llbracket i \rrbracket_i^\sharp : (\mathbb{L} \times \mathbb{T}^*) \to \mathbb{S}^\sharp \to \mathbb{S}^\sharp$ (Section III-B), references $\llbracket r \rrbracket_r^\sharp : \mathbb{E}^\sharp \to \mathbb{T}^\sharp$ (Section III-C), and expressions $\llbracket e \rrbracket_e^\sharp : \mathbb{E}^\sharp \to \mathbb{T}^\sharp$ (Section III-D).

A. Abstract States: $\mathbb{S}^\sharp$

Before defining abstract states, we first extend types as follows:

$$\mathbb{T} \ni \tau ::= \cdots \mid f \mid c \mid b \mid s \mid \text{?} \mid \text{normal}(\tau) \mid \text{abrupt}$$

We add types for functions f and constants c, Boolean values b and String values s to precisely handle the control flows of branches and field accesses, respectively, the absent type ? to represent the absence of variables, and $\text{normal}(\tau)$ for normal completions whose Value fields have type τ and abrupt for abrupt completions to enhance the analysis precision.

Using the extended types, we define abstract states with flow-sensitivity and type-sensitivity for arguments:

Abstract States	$d^\sharp \in \mathbb{S}^\sharp = \mathbb{M} \times \mathbb{R}$
Result Maps	$m \in \mathbb{M} = \mathbb{L} \times \mathbb{T}^* \to \mathbb{E}^\sharp$
Return Point Maps	$r \in \mathbb{R} = \mathbb{F} \times \mathbb{T}^* \to \mathcal{P}(\mathbb{L} \times \mathbb{T}^* \times \mathbb{X})$
Abstract Environments	$\sigma^\sharp \in \mathbb{E}^\sharp = \mathbb{X} \to \mathbb{T}^\sharp$
Abstract Types	$\tau^\sharp \in \mathbb{T}^\sharp = \mathcal{P}(\mathbb{T})$

An abstract state $d^\sharp \in \mathbb{S}^\sharp$ is a pair of a result map and a return point map. A result map $m \in \mathbb{M}$ represents an abstract environment for each flow- and type-sensitive view, and a return point map $r \in \mathbb{R}$ represents possible return points of each function with a type-sensitive context; each return point consists of a view for the caller function and a variable that represents the return value. An abstract environment $\sigma^\sharp \in \mathbb{E}^\sharp$ represents possible types for variables, and $\sigma^\sharp(\text{x}) = \{\text{?}\}$ when x is not defined in $\sigma^\sharp$. An abstract type $\tau^\sharp \in \mathbb{T}^\sharp$ is a set of types. We define the join operator $\sqcup$, the meet operator $\sqcap$, and the partial order $\sqsubseteq$ for most of abstract domains in a point-wise manner, and define the operators for types with a normalization function norm because of their subtype relations:

$$\tau_0^\sharp \sqcup \tau_1^\sharp = \text{norm}(\tau_0^\sharp \cup \tau_1^\sharp)$$
$$\tau_0^\sharp \sqcap \tau_1^\sharp = \text{norm}(\{\tau_0 \in \tau_0^\sharp \mid \{\tau_0\} \sqsubseteq \tau_1^\sharp\} \cup \{\tau_1 \in \tau_1^\sharp \mid \{\tau_1\} \sqsubseteq \tau_0^\sharp\})$$
$$\tau_0^\sharp \sqsubseteq \tau_1^\sharp \Leftrightarrow \forall \tau_0 \in \tau_0^\sharp. \exists \tau_1 \in \text{norm}(\tau_1^\sharp). \text{ s.t. } \tau_0 <: \tau_1$$

where $\text{norm}(\tau^\sharp) = \{\tau \mid \tau \in \tau^\sharp \wedge \nexists \tau' \in \tau^\sharp \setminus \{\tau\}. \text{ s.t. } \tau <: \tau'\}$. Then, we define the abstract semantics $[\![P]\!]^\sharp$ of a program P as the least fixpoint of the abstract transfer $F^\sharp : \mathbb{S}^\sharp \to \mathbb{S}^\sharp$:

$$[\![P]\!]^\sharp = \lim_{n \to \infty}(F^\sharp)^n(d_\iota^\sharp)$$
$$F^\sharp(d^\sharp) = d^\sharp \sqcup \left(\bigsqcup_{(\ell, \overline{\tau}) \in \text{Domain}(m)} [\![\text{inst}(\ell)]\!]_i^\sharp(\ell, \overline{\tau})(d^\sharp) \right)$$

where $d^\sharp = (m, \_)$ and $d_\iota^\sharp$ denotes the initial abstract state.

B. Instructions: $[\![i]\!]_i^\sharp : (\mathbb{L} \times \mathbb{T}^*) \to \mathbb{S}^\sharp \to \mathbb{S}^\sharp$

- **Variable Declarations:**

$$[\![\text{let } \text{x} = e]\!]_i^\sharp(\ell, \overline{\tau})(d^\sharp) = (\{(\text{next}(\ell), \overline{\tau}) \mapsto \sigma_\text{x}^\sharp\}, \varnothing)$$

  where
$$d^\sharp = (m, \_) \wedge \sigma^\sharp = m(\ell, \overline{\tau}) \wedge$$
$$\sigma_\text{x}^\sharp = \sigma^\sharp[\text{x} \mapsto [\![e]\!]_e^\sharp(\sigma^\sharp)]$$

- **Function Calls:**

$$[\![\text{x} = (e\ e_1 \cdots e_n)]\!]_i^\sharp(\ell, \overline{\tau})(d^\sharp) = (m', r')$$

where
$$d^\sharp = (m, \_) \wedge \sigma^\sharp = m(\ell, \overline{\tau}) \wedge$$
$$\tau^\sharp = [\![e]\!]_e^\sharp(\sigma^\sharp) \wedge$$
$$\tau_1^\sharp = [\![e_1]\!]_e^\sharp(\sigma^\sharp) \wedge \cdots \wedge \tau_n^\sharp = [\![e_n]\!]_e^\sharp(\sigma^\sharp) \wedge$$
$$T' = \{\text{up}([\tau_1, \cdots, \tau_n]) \mid \tau_1 \in \tau_1^\sharp \wedge \cdots \wedge \tau_n \in \tau_n^\sharp\} \wedge$$
$$f = \text{def } \text{f}(\text{p}_1, \cdots, [\cdots, \text{p}_{k_f}]). \ \ell_f \wedge$$
$$\sigma_{f,\overline{\tau}'}^\sharp = [\text{p}_1 \mapsto \{\overline{\tau}'[1]\}, \cdots, \text{p}_{k_f} \mapsto \{\overline{\tau}'[k_f]\}] \wedge$$
$$m' = \{(\ell_f, \overline{\tau}') \mapsto \sigma_{f,\overline{\tau}'}^\sharp \mid f \in \tau^\sharp \wedge \overline{\tau}' \in T'\} \wedge$$
$$r' = \{(f, \overline{\tau}') \mapsto \{(\text{next}(\ell), \overline{\tau}, \text{x})\} \mid f \in \tau^\sharp \wedge \overline{\tau}' \in T'\}$$

- **Returns:**

$$[\![\text{return } e]\!]_i^\sharp(\ell, \overline{\tau})(d^\sharp) = (m', \varnothing)$$

where
$$d^\sharp = (m, r) \wedge \sigma^\sharp = m(\ell, \overline{\tau}) \wedge$$
$$R = r(\text{func}(\ell), \overline{\tau}) \wedge$$
$$m' = \{(\ell_r, \overline{\tau}_r) \mapsto \sigma_r^\sharp \mid (\ell_r, \overline{\tau}_r, \text{x}) \in R \wedge$$
$$\sigma_r^\sharp = m(\ell_r, \overline{\tau}_r)[\text{x} \mapsto [\![e]\!]_e^\sharp(\sigma^\sharp)]\}$$

- **Assertions:**

$$[\![\text{assert } e]\!]_i^\sharp(\ell, \overline{\tau})(d^\sharp) = (m', \varnothing)$$

where
$$d^\sharp = (m, \_) \wedge \sigma^\sharp = m(\ell, \overline{\tau}) \wedge$$
$$m' = \{(\text{next}(\ell), \overline{\tau}) \mapsto \text{pass}(e, \#\text{t})(\sigma^\sharp)\}$$

- **Branches:**

$$[\![\text{if } e\ \ell_t\ \ell_f]\!]_i^\sharp(\ell, \overline{\tau})(d^\sharp) = (m', \varnothing)$$

where
$$d^\sharp = (m, \_) \wedge \sigma^\sharp = m(\ell, \overline{\tau}) \wedge$$
$$m' = \left\{ \begin{array}{l} (\ell_t, \overline{\tau}) \mapsto \text{pass}(e, \#\text{t})(\sigma^\sharp), \\ (\ell_f, \overline{\tau}) \mapsto \text{pass}(e, \#\text{f})(\sigma^\sharp) \end{array} \right\}$$

- **Variable Updates:**

$$[\![\text{x} = e]\!]_i^\sharp(\ell, \overline{\tau})(d^\sharp) = (\{(\text{next}(\ell), \overline{\tau}) \mapsto d_\text{x}^\sharp\}, \varnothing)$$

where
$$d^\sharp = (m, \_) \wedge \sigma^\sharp = m(\ell, \overline{\tau}) \wedge$$
$$d_\text{x}^\sharp = \sigma^\sharp[\text{x} \mapsto [\![e]\!]_e^\sharp(\sigma^\sharp)]$$

- **Field Updates:**

$$[\![r[e_0] = e_1]\!]_i^\sharp(\ell, \overline{\tau})(d^\sharp) = (\{(\text{next}(\ell), \overline{\tau}) \mapsto \sigma^\sharp\}, \varnothing)$$

where
$$d^\sharp = (m, \_) \wedge \sigma^\sharp = m(\ell, \overline{\tau})$$

To avoid the explosion of type-sensitive views, we upcast the argument type before function calls with the following function:

$$\text{up}(\tau) = \left\{ \begin{array}{ll} \text{normal}(\text{up}(\tau')) & \text{if } \tau = \text{normal}(\tau') \\ [\text{up}(\tau')] & \text{if } \tau = [\tau'] \\ \text{str} & \text{if } \tau = s \\ \text{bool} & \text{if } \tau = b \\ \tau & \text{otherwise} \end{array} \right.$$

and $\dot{\text{up}}$ denotes a point-wise extension of up for type sequences. For branches and assertions, we use the following pass function to prevent infeasible control flows:

$$\text{pass}(e, b)(\sigma^\sharp) = \begin{cases} \text{refine}(e, b)(\sigma^\sharp) & \text{if } \{\#\text{t}\} \sqsubseteq [\![e]\!]_e^\sharp(\sigma^\sharp) \\ \varnothing & \text{otherwise} \end{cases}$$

where refine is a function that performs *condition-based refinement* of the type analysis for the modified IR_{ES} to enhance the analysis precision. It prunes out infeasible parts in abstract environments using the conditions of branches and assertions. We formally define the refine function as follows:

$$\begin{aligned}
\text{refine}(!e, b)(\sigma^\sharp) &= \text{refine}(e, \neg b)(\sigma^\sharp) \\
\text{refine}(e_0 \mathrel{||} e_1, b)(\sigma^\sharp) &= \begin{cases} \sigma_0^\sharp \sqcup \sigma_1^\sharp & \text{if } b \\ \sigma_0^\sharp \sqcap \sigma_1^\sharp & \text{if } \neg b \end{cases} \\
\text{refine}(e_0 \mathrel{\&\&} e_1, b)(\sigma^\sharp) &= \begin{cases} \sigma_0^\sharp \sqcap \sigma_1^\sharp & \text{if } b \\ \sigma_0^\sharp \sqcup \sigma_1^\sharp & \text{if } \neg b \end{cases} \\
\text{refine}(\text{x.Type} == c_{\text{normal}}, \#\text{t})(\sigma^\sharp) &= \sigma^\sharp[\text{x} \mapsto \tau_\text{x}^\sharp \sqcap \text{normal}(\mathbb{T})] \\
\text{refine}(\text{x.Type} == c_{\text{normal}}, \#\text{f})(\sigma^\sharp) &= \sigma^\sharp[\text{x} \mapsto \tau_\text{x}^\sharp \sqcap \{\text{abrupt}\}] \\
\text{refine}(\text{x} == e, \#\text{t})(\sigma^\sharp) &= \sigma^\sharp[\text{x} \mapsto \tau_\text{x}^\sharp \sqcap \tau_e^\sharp] \\
\text{refine}(\text{x} == e, \#\text{f})(\sigma^\sharp) &= \sigma^\sharp[\text{x} \mapsto \tau_\text{x}^\sharp \setminus \lfloor \tau_e^\sharp \rfloor] \\
\text{refine}(\text{x} : \tau, \#\text{t})(\sigma^\sharp) &= \sigma^\sharp[\text{x} \mapsto \tau_\text{x}^\sharp \sqcap \{\tau\}] \\
\text{refine}(\text{x} : \tau, \#\text{f})(\sigma^\sharp) &= \sigma^\sharp[\text{x} \mapsto \tau_\text{x}^\sharp \setminus \{\tau' \mid \tau' <: \tau\}] \\
\text{refine}(e, b)(\sigma^\sharp) &= \sigma^\sharp
\end{aligned}$$

where $\sigma_j^\sharp = \text{refine}(e_j, b)(\sigma^\sharp)$ for $j = 0, 1$, $\tau_e^\sharp = [\![e]\!]_e^\sharp(\sigma^\sharp)$, and $\lfloor \tau^\sharp \rfloor$ returns $\{\tau\}$ if $\tau^\sharp$ denotes a singleton type τ, or returns $\varnothing$, otherwise.

C. References: $[\![r]\!]_r^\sharp : \mathbb{E}^\sharp \to \mathbb{T}^\sharp$

- Variable Lookups:

$$[\![\text{x}]\!]_r^\sharp(\sigma^\sharp) = \sigma^\sharp(\text{x})$$

- Field Lookups:

$$[\![r\text{[}e\text{]}]\!]_r^\sharp(\sigma^\sharp) = \{\tau[v] \mid \tau \in [\![r]\!]_r^\sharp(\sigma^\sharp) \wedge v \in [\![e]\!]_e^\sharp(\sigma^\sharp)\}$$

where $\tau[v]$ denotes the access of field v for a type τ.

D. Expressions: $[\![e]\!]_e^\sharp : \mathbb{E}^\sharp \to \mathbb{T}^\sharp$

- Completion Records:

$$\begin{aligned}
&[\![\text{Completion } \{\cdots, \text{Type} : e_0, \text{Value} : e_1, \cdots\}]\!]_e^\sharp(\sigma^\sharp) \\
&= \begin{cases} \{\text{normal}(\tau) \mid \tau \in [\![e_1]\!]_e^\sharp(\sigma^\sharp)\} & \text{if } [\![e_0]\!]_e^\sharp = c_{\text{normal}} \\ \{\text{abrupt}\} & \text{otherwise} \end{cases}
\end{aligned}$$

- Records:

$$[\![t \text{ } \{\cdots\}]\!]_e^\sharp(\sigma^\sharp) = \{t\}$$

- Lists:

$$\begin{aligned}
{[\![}\text{[]}{]\!]}_e^\sharp(\sigma^\sharp) &= \text{[]} \\
{[\![}\text{[}e_1, \cdots, e_n\text{]}{]\!]}_e^\sharp(\sigma^\sharp) &= \{\text{[}\tau\text{]} \mid \tau \in \bigsqcup_{1 \leq i \leq n} [\![e_i]\!]_e^\sharp(\sigma^\sharp)\}
\end{aligned}$$

- Type Checks:

$$[\![e : \tau]\!]_e^\sharp(\sigma^\sharp) = \{\tau' <: \tau \mid \tau' \in [\![e]\!]_e^\sharp(\sigma^\sharp)\}$$

- Existence Checks:

$$[\![r?]\!]_e^\sharp(\sigma^\sharp) = \{\tau \neq \text{?} \mid \tau \in [\![e]\!]_e^\sharp(\sigma^\sharp)\}$$

- Binary Operations:

$$[\![e_0 \oplus e_1]\!]_e^\sharp(\sigma^\sharp) = \{\tau_0 \oplus^\sharp \tau_1 \mid \tau_0 \in \tau_0^\sharp \wedge \tau_1 \in \tau_1^\sharp\}$$

where

$$\tau_0^\sharp = [\![e_0]\!]_e^\sharp(\sigma^\sharp) \wedge \tau_1^\sharp = [\![e_1]\!]_e^\sharp(\sigma^\sharp)$$

- Unary Operations:

$$[\![\ominus e]\!]_e^\sharp(\sigma^\sharp) = \{\ominus^\sharp \tau \mid \tau \in [\![e]\!]_e^\sharp(\sigma^\sharp)\}$$

- References:

$$[\![r]\!]_e^\sharp(\sigma^\sharp) = [\![r]\!]_r^\sharp(\sigma^\sharp) \setminus \{\text{?}\}$$

- Constants:

$$[\![c]\!]_e^\sharp(\sigma^\sharp) = c$$

- Primitives:

$$[\![p]\!]_e^\sharp(\sigma^\sharp) = \begin{cases} \text{num} & \text{if } p = n \\ \text{bigint} & \text{if } p = j \\ \text{symbol} & \text{if } p = @s \\ p & \text{otherwise} \end{cases}$$