Type Analysis for a Modified IR_{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_{ES}, an untyped intermediate representation for ECMAScript. Moreover, we formally define type analysis for the modified IR_{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_{ES} as follows:

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

A modified IR_{ES} program $P = (\texttt{func}, \texttt{inst}, \texttt{next})$ consists of three mappings; $\texttt{func} : \mathbb{L} \to \mathbb{F}$ maps labels to their functions, $\texttt{inst} : \mathbb{L} \to \mathbb{I}$ maps labels to their instructions, and $\texttt{next} : \mathbb{L} \to \mathbb{L}$ maps labels to their next labels, where a label $\ell \in \mathbb{L}$ denotes a program point. A function $\texttt{def f}(\texttt{x}^*, \texttt{[y}^*\texttt{]}).\ \ell \in \mathbb{F}$ consists of its name $\texttt{f}$, normal parameters $\texttt{x}^*$, optional parameters $\texttt{y}^*$, 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 $\texttt{next}$. A reference r is a variable $\texttt{x}$ or a field access $r\texttt{[}e\texttt{]}$. We write $r.\texttt{f}$ to briefly represent $r\texttt{["f"]}$. 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 `undefined`, `null`, a `Boolean` b, a `Number` n, a `BigInt` j, a `String` s, or a `Symbol` $\texttt{@}s$.

A type $\tau \in \mathbb{T}$ is either a nominal type t, an empty list type `[]`, a parametric list type $\texttt{[}\tau\texttt{]}$, a JavaScript type `js`, a primitive type `prim`, a numeric type `numeric`, `num`, `bigint`, `str`, or `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_{ES}. We will define states $\mathbb{S}$ (Section II-A), and then define a denotational semantics of the modified IR_{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:

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

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:

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

where

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

- Function Calls:

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

where

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

- Assertions:

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

- Branches:

$$[\![\texttt{if}\ e\ \ell_{\texttt{t}}\ \ell_{\texttt{f}}]\!]_i(d) = \begin{cases} (\ell_{\texttt{t}}, \overline{\kappa}, h, \sigma) & \text{if } v = \texttt{\#t} \\ (\ell_{\texttt{f}}, \overline{\kappa}, h, \sigma) & \text{if } v = \texttt{\#f} \end{cases}$$

where

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

- Returns:

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

where

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

- Variable Updates:

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

where

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

- Field Updates:

$$[\![r\texttt{[}e_0\texttt{]} = e]\!]_i(d_1) = (\texttt{next}(\ell), \overline{\kappa}, h[a \mapsto o'], \sigma)$$

where

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

C. References: $[\![r]\!]_r : \mathbb{S} \to \mathbb{S} \times \mathbb{V}$

- Variable Lookups:

$$[\![\texttt{x}]\!]_r(d) = (d, \sigma(\texttt{x}))$$

where

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

- Field Lookups:

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

where

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

D. Expressions: $[\![e]\!]_e : \mathbb{S} \to \mathbb{S} \times \mathbb{V}$

- Records:

$$[\![t\ \texttt{\{}\texttt{x}_1 : e_1, \cdots, \texttt{x}_n : e_n\texttt{\}}]\!]_e(d) = (d', a)$$

where

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

- Lists:

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

where

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

- Type Checks:

$$[\![e : \tau]\!]_e(d) = (d', b)$$

where

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

- Variable Existence Checks:

$$[\![\texttt{x?}]\!]_e(d) = (d, b)$$

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:

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

where

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

- Binary Operations:

$$[\![e \oplus e]\!]_e(d) = (d'', v_0 \oplus v_1)$$

where

$$[\![e_0]\!]_e(d) = (d', v_0) \wedge [\![e_1]\!]_e(d') = (d'', v_1)$$

- Unary Operations:

$$[\![\ominus\ e]\!]_e(d) = (d', \ominus\ v)$$

where

$$[\![e]\!]_e(d) = (d', v)$$

- References:

$$[\![r]\!]_e(d) = [\![r]\!]_r(d)$$

- Constants:

$$[\![c]\!]_e(d) = (d, c)$$

- Primitives:

$$[\![p]\!]_e(d) = (d, p)$$

III. Type Analysis

We design a type analysis for the modified IR_{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_{ES} for instructions $[\![i]\!]_i^\sharp : (\mathbb{L} \times \mathbb{T}^*) \to \mathbb{S}^\sharp \to \mathbb{S}^\sharp$ (Section III-B), references $[\![r]\!]_r^\sharp : \mathbb{E}^\sharp \to \mathbb{T}^\sharp$ (Section III-C), and expressions $[\![e]\!]_e^\sharp : \mathbb{E}^\sharp \to \mathbb{T}^\sharp$ (Section III-D).

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

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

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

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 $\texttt{normal}(\tau)$ 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 $\sigma^\sharp(\texttt{x}) = \{\texttt{?}\}$ 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:

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

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

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

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:

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

where

$$\begin{array}{l}
d^\sharp = (m, _) \wedge \sigma^\sharp = m(\ell,\overline{\tau}) \wedge \\
\sigma_{\texttt{x}}^\sharp = \sigma^\sharp[\texttt{x} \mapsto [\![e]\!]_e^\sharp(\sigma^\sharp)]
\end{array}$$

- Function Calls:

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

where

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

- Returns:

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

where

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

- Assertions:

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

where

$$\begin{array}{l}
d^\sharp = (m, _) \wedge \sigma^\sharp = m(\ell,\overline{\tau}) \wedge \\
m' = \{(\texttt{next}(\ell), \overline{\tau}) \mapsto \texttt{pass}(e, \texttt{\#t})(\sigma^\sharp)\}
\end{array}$$

- Branches:

$$[\![\texttt{if } e\ \ell_{\texttt{t}}\ \ell_{\texttt{f}}]\!]_i^\sharp(\ell,\overline{\tau})(d^\sharp) = (m', \varnothing)$$

where

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

- Variable Updates:

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

where

$$\begin{array}{l}
d^\sharp = (m, _) \wedge \sigma^\sharp = m(\ell,\overline{\tau}) \wedge \\
d_{\texttt{x}}^\sharp = \sigma^\sharp[\texttt{x} \mapsto [\![e]\!]_e^\sharp(\sigma^\sharp)]
\end{array}$$

- Field Updates:

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

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:

$$\texttt{up}(\tau) = \begin{cases} \texttt{normal}(\texttt{up}(\tau')) & \text{if } \tau = \texttt{normal}(\tau') \\ \texttt{[}\texttt{up}(\tau')\texttt{]} & \text{if } \tau = \texttt{[}\tau'\texttt{]} \\ \texttt{str} & \text{if } \tau = s \\ \texttt{bool} & \text{if } \tau = b \\ \tau & \text{otherwise} \end{cases}$$

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:

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

where refine is a funcition 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:

$$\begin{aligned}
\texttt{refine}(!e, b)(\sigma^\sharp) &= \texttt{refine}(e, \neg b)(\sigma^\sharp) \\
\texttt{refine}(e_0 \;\texttt{||}\; e_1, b)(\sigma^\sharp) &= \begin{cases} \sigma^\sharp_0 \sqcup \sigma^\sharp_1 & \text{if } b \\ \sigma^\sharp_0 \sqcap \sigma^\sharp_1 & \text{if } \neg b \end{cases} \\
\texttt{refine}(e_0 \;\texttt{\&\&}\; e_1, b)(\sigma^\sharp) &= \begin{cases} \sigma^\sharp_0 \sqcap \sigma^\sharp_1 & \text{if } b \\ \sigma^\sharp_0 \sqcup \sigma^\sharp_1 & \text{if } \neg b \end{cases} \\
\texttt{refine}(\texttt{x.Type == } c_{\texttt{normal}}, \texttt{\#t})(\sigma^\sharp) &= \sigma^\sharp[\texttt{x} \mapsto \tau^\sharp_{\texttt{x}} \cap \texttt{normal}(\mathbb{T})] \\
\texttt{refine}(\texttt{x.Type == } c_{\texttt{normal}}, \texttt{\#f})(\sigma^\sharp) &= \sigma^\sharp[\texttt{x} \mapsto \tau^\sharp_{\texttt{x}} \cap \{\texttt{abrupt}\}] \\
\texttt{refine}(\texttt{x == } e, \texttt{\#t})(\sigma^\sharp) &= \sigma^\sharp[\texttt{x} \mapsto \tau^\sharp_{\texttt{x}} \sqcap \tau^\sharp_e] \\
\texttt{refine}(\texttt{x == } e, \texttt{\#f})(\sigma^\sharp) &= \sigma^\sharp[\texttt{x} \mapsto \tau^\sharp_{\texttt{x}} \setminus \lfloor \tau^\sharp_e \rfloor] \\
\texttt{refine}(\texttt{x} : \tau, \texttt{\#t})(\sigma^\sharp) &= \sigma^\sharp[\texttt{x} \mapsto \tau^\sharp_{\texttt{x}} \sqcap \{\tau\}] \\
\texttt{refine}(\texttt{x} : \tau, \texttt{\#f})(\sigma^\sharp) &= \sigma^\sharp[\texttt{x} \mapsto \tau^\sharp_{\texttt{x}} \setminus \{\tau' \mid \tau' <: \tau\}] \\
\texttt{refine}(e, b)(\sigma^\sharp) &= \sigma^\sharp
\end{aligned}$$

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

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

- Variable Lookups:

$$[\![\texttt{x}]\!]^\sharp_r(\sigma^\sharp) = \sigma^\sharp(\texttt{x})$$

- Field Lookups:

$$[\![r\,\texttt{[}e\texttt{]}]\!]^\sharp_r(\sigma^\sharp) = \{\tau[v] \mid \tau \in [\![r]\!]^\sharp_r(\sigma^\sharp) \wedge v \in [\![e]\!]^\sharp_e(\sigma^\sharp)\}$$

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

- Completion Records:

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

- Records:

$$[\![t \; \texttt{\{} \cdots \texttt{\}}]\!]^\sharp_e(\sigma^\sharp) = \{t\}$$

- Lists:

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

- Type Checks:

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

- Existence Checks:

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

- Binary Operations:

$$[\![e_0 \oplus e_1]\!]^\sharp_e(\sigma^\sharp) = \{\tau_0 \oplus^\sharp \tau_1 \mid \tau_0 \in \tau^\sharp_0 \wedge \tau_1 \in \tau^\sharp_1\}$$

where

$$\tau^\sharp_0 = [\![e_0]\!]^\sharp_e(\sigma^\sharp) \wedge \tau^\sharp_1 = [\![e_1]\!]^\sharp_e(\sigma^\sharp)$$

- Unary Operations:

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

- References:

$$[\![r]\!]^\sharp_e(\sigma^\sharp) = [\![r]\!]^\sharp_r(\sigma^\sharp) \setminus \{\texttt{?}\}$$

- Constants:

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

- Primitives:

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