

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:

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

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

A type $\tau \in \mathbb{T}$ is either a nominal type t , an empty list type $[]$, a parametric list type $[\tau]$, 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 $\llbracket i \rrbracket_i : \mathbb{S} \rightarrow \mathbb{S}$ (Section II-B), references $\llbracket r \rrbracket_r : \mathbb{S} \rightarrow \mathbb{S} \times \mathbb{V}$ (Section II-C), and expressions $\llbracket e \rrbracket_e : \mathbb{S} \rightarrow \mathbb{S} \times \mathbb{V}$ (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	$h \in \mathbb{H} = \mathbb{A} \rightarrow \mathbb{O}$
Addresses	$a \in \mathbb{A}$
Objects	$o \in \mathbb{O} = (\mathbb{T}_t \times (\mathbb{V}_s \rightarrow \mathbb{V})) \uplus \mathbb{V}^*$
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: $\llbracket i \rrbracket_i : \mathbb{S} \rightarrow \mathbb{S}$

• Variable Declarations:

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

where

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

• Function Calls:

$$\llbracket x = (e_0 e_1 \cdots e_n) \rrbracket_i(d) = (\ell_f, \kappa :: \bar{\kappa}, h, \sigma')$$

where

$$\begin{aligned} \llbracket e_0 \rrbracket_e(d) &= (d_0, \text{def } f(p_1, \dots, p_m) \ell_f \wedge \\ \llbracket e_1 \rrbracket_e(d_0) &= (d_1, v_1) \wedge \cdots \wedge \llbracket e_n \rrbracket_e(d_{n-1}) = (d_n, v_n) \wedge \\ d_n &= (\ell, \bar{\kappa}, h, \sigma) \wedge k = \min(n, m) \wedge \\ \sigma' &= [p_1 \mapsto v_1, \dots, p_k \mapsto v_k] \wedge \kappa = (\text{next}(\ell), \sigma, x) \end{aligned}$$

• Assertions:

$$\llbracket \text{assert } e \rrbracket_i(d) = d' \quad \text{if } \llbracket e \rrbracket_e(d) = (d', \#t)$$

- Branches:

$$\llbracket \text{if } e \text{ then } l_{\text{t}} \text{ else } l_{\text{f}} \rrbracket_i(d) = \begin{cases} (l_{\text{t}}, \bar{\kappa}, h, \sigma) & \text{if } v = \#_{\text{t}} \\ (l_{\text{f}}, \bar{\kappa}, h, \sigma) & \text{if } v = \#_{\text{f}} \end{cases}$$

where

$$\llbracket e \rrbracket_e(d) = ((l_{\text{t}}, \bar{\kappa}, h, \sigma), v)$$

- Returns:

$$\llbracket \text{return } e \rrbracket_i(d) = (l, \bar{\kappa}, h, \sigma[x \mapsto v])$$

where

$$\llbracket e \rrbracket_e(d) = ((_, (l, \sigma, x) :: \bar{\kappa}, h, _), v)$$

- Variable Updates:

$$\llbracket x = e \rrbracket_i(d) = (\text{next}(l), \bar{\kappa}, h, \sigma[x \mapsto v])$$

where

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

- Field Updates:

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

where

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

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

- Variable Lookups:

$$\llbracket x \rrbracket_r(d) = (d, \sigma(x))$$

where

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

- Field Lookups:

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

where

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

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

- Records:

$$\llbracket t \{x_1 : e_1, \dots, x_n : e_n\} \rrbracket_e(d) = (d', a)$$

where

$$\begin{aligned} \llbracket e_1 \rrbracket_e(d) &= (d_1, v_1) \wedge \dots \wedge \llbracket e_n \rrbracket_e(d_{n-1}) = (d_n, v_n) \wedge \\ d_n &= (l, \bar{\kappa}, h, \sigma) \wedge \text{fs} = [x_1 \mapsto v_1, \dots, x_n \mapsto v_n] \\ a &\notin \text{Domain}(h) \wedge d' = (l, \bar{\kappa}, h[a \mapsto (t, \text{fs})], \sigma) \end{aligned}$$

- Lists:

$$\llbracket [e_1, \dots, e_n] \rrbracket_e(d) = (d', a)$$

where

$$\begin{aligned} \llbracket e_1 \rrbracket_e(d) &= (d_1, v_1) \wedge \dots \wedge \llbracket e_n \rrbracket_e(d_{n-1}) = (d_n, v_n) \wedge \\ d_n &= (l, \bar{\kappa}, h, \sigma) \wedge a \notin \text{Domain}(h) \wedge \\ d' &= (l, \bar{\kappa}, h[a \mapsto [v_1, \dots, v_n]], \sigma) \end{aligned}$$

- Type Checks:

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

where

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

- Variable Existence Checks:

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

where

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

- Field Existence Checks:

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

where

$$\begin{aligned} \llbracket r \rrbracket_e(d) &= (d', a) \wedge \llbracket e \rrbracket_e(d') = (d'', v) \wedge \\ d'' &= (l, \bar{\kappa}, h, \sigma) \wedge o = h(a) \wedge \\ b &= \begin{cases} \#_{\text{t}} & \text{if } o = (t, \text{fs}) \wedge v = s \wedge s \in \text{Domain}(\text{fs}) \\ \#_{\text{t}} & \text{if } o = [v'_1, \dots, v'_m] \wedge v = n \wedge 1 \leq n \leq m \\ \#_{\text{f}} & \text{otherwise} \end{cases} \end{aligned}$$

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

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

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

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

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	$m \in \mathbb{M} = \mathbb{L} \times \mathbb{T}^* \rightarrow \mathbb{E}^\sharp$
Return Point Maps	$r \in \mathbb{R} = \mathbb{F} \times \mathbb{T}^* \rightarrow \mathcal{P}(\mathbb{L} \times \mathbb{T}^* \times \mathbb{X})$
Abstract Environments	$\sigma^\sharp \in \mathbb{E}^\sharp = \mathbb{X} \rightarrow \mathbb{T}^\sharp$
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 $\sigma^\sharp(x) = \{?\}$ when x is not defined in σ^\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:

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

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 $\llbracket P \rrbracket^\sharp$ of a program P as the least fixpoint of the abstract transfer $F^\sharp : \mathbb{S}^\sharp \rightarrow \mathbb{S}^\sharp$:

$$\begin{aligned} \llbracket P \rrbracket^\sharp &= \lim_{n \rightarrow \infty} (F^\sharp)^n(d_i^\sharp) \\ F^\sharp(d^\sharp) &= d^\sharp \sqcup \left(\bigsqcup_{(l, \bar{\tau}) \in \text{Domain}(m)} \llbracket \text{inst}(l) \rrbracket_i^\sharp(l, \bar{\tau})(d^\sharp) \right) \end{aligned}$$

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

B. Instructions: $\llbracket i \rrbracket_i^\sharp : (\mathbb{L} \times \mathbb{T}^*) \rightarrow \mathbb{S}^\sharp \rightarrow \mathbb{S}^\sharp$

• Variable Declarations:

$$\llbracket \text{let } x = e \rrbracket_i^\sharp(l, \bar{\tau})(d^\sharp) = (\{(next(l), \bar{\tau}) \mapsto \sigma_x^\sharp\}, \emptyset)$$

where

$$\begin{aligned} d^\sharp &= (m, _) \wedge \sigma^\sharp = m(l, \bar{\tau}) \wedge \\ \sigma_x^\sharp &= \sigma^\sharp[x \mapsto \llbracket e \rrbracket_e^\sharp(\sigma^\sharp)] \end{aligned}$$

• Function Calls:

$$\llbracket x = (e_1 \dots e_n) \rrbracket_i^\sharp(l, \bar{\tau})(d^\sharp) = (m', r')$$

where

$$\begin{aligned} d^\sharp &= (m, _) \wedge \sigma^\sharp = m(l, \bar{\tau}) \wedge \\ \tau^\sharp &= \llbracket e \rrbracket_e^\sharp(\sigma^\sharp) \wedge \\ \tau_1^\sharp &= \llbracket e_1 \rrbracket_e^\sharp(\sigma^\sharp) \wedge \dots \wedge \tau_n^\sharp = \llbracket e_n \rrbracket_e^\sharp(\sigma^\sharp) \wedge \\ T' &= \{\text{up}([\tau_1, \dots, \tau_n]) \mid \tau_1 \in \tau_1^\sharp \wedge \dots \wedge \tau_n \in \tau_n^\sharp\} \wedge \\ f &= \text{def } f(p_1, \dots, [\dots, p_{k_f}]) \ell_f \wedge \\ \sigma_{f, \bar{\tau}'}^\sharp &= [p_1 \mapsto \{\bar{\tau}'[1]\}, \dots, p_{k_f} \mapsto \{\bar{\tau}'[k_f]\}] \wedge \\ m' &= \{(\ell_f, \bar{\tau}') \mapsto \sigma_{f, \bar{\tau}'}^\sharp \mid f \in \tau^\sharp \wedge \bar{\tau}' \in T'\} \wedge \\ r' &= \{(f, \bar{\tau}') \mapsto \{(next(l), \bar{\tau}, x)\} \mid f \in \tau^\sharp \wedge \bar{\tau}' \in T'\} \end{aligned}$$

• Returns:

$$\llbracket \text{return } e \rrbracket_i^\sharp(l, \bar{\tau})(d^\sharp) = (m', \emptyset)$$

where

$$\begin{aligned} d^\sharp &= (m, r) \wedge \sigma^\sharp = m(l, \bar{\tau}) \wedge \\ R &= r(\text{func}(l), \bar{\tau}) \wedge \\ m' &= \{(\ell_r, \bar{\tau}_r) \mapsto \sigma_r^\sharp \mid (\ell_r, \bar{\tau}_r, x) \in R \wedge \\ &\quad \sigma_r^\sharp = m(\ell_r, \bar{\tau}_r)[x \mapsto \llbracket e \rrbracket_e^\sharp(\sigma^\sharp)]\} \end{aligned}$$

• Assertions:

$$\llbracket \text{assert } e \rrbracket_i^\sharp(l, \bar{\tau})(d^\sharp) = (m', \emptyset)$$

where

$$\begin{aligned} d^\sharp &= (m, _) \wedge \sigma^\sharp = m(l, \bar{\tau}) \wedge \\ m' &= \{(next(l), \bar{\tau}) \mapsto \text{pass}(e, \#t)(\sigma^\sharp)\} \end{aligned}$$

• Branches:

$$\llbracket \text{if } e \ell_t \ell_f \rrbracket_i^\sharp(l, \bar{\tau})(d^\sharp) = (m', \emptyset)$$

where

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

• Variable Updates:

$$\llbracket x = e \rrbracket_i^\sharp(l, \bar{\tau})(d^\sharp) = (\{(next(l), \bar{\tau}) \mapsto d_x^\sharp\}, \emptyset)$$

where

$$\begin{aligned} d^\sharp &= (m, _) \wedge \sigma^\sharp = m(l, \bar{\tau}) \wedge \\ d_x^\sharp &= \sigma^\sharp[x \mapsto \llbracket e \rrbracket_e^\sharp(\sigma^\sharp)] \end{aligned}$$

• Field Updates:

$$\llbracket r[e_0] = e_1 \rrbracket_i^\sharp(l, \bar{\tau})(d^\sharp) = (\{(next(l), \bar{\tau}) \mapsto \sigma^\sharp\}, \emptyset)$$

where

$$d^\sharp = (m, _) \wedge \sigma^\sharp = m(l, \bar{\tau})$$

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

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

and 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:

$$\text{pass}(e, b)(\sigma^\#) = \begin{cases} \text{refine}(e, b)(\sigma^\#) & \text{if } \{\#t\} \sqsubseteq \llbracket e \rrbracket_e^\#(\sigma^\#) \\ \emptyset & \text{otherwise} \end{cases}$$

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:

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

where $\sigma_j^\# = \text{refine}(e_j, b)(\sigma^\#)$ for $j = 0, 1$, $\tau_e^\# = \llbracket e \rrbracket_e^\#(\sigma^\#)$, and $\lfloor \tau^\# \rfloor$ returns $\{\tau\}$ if $\tau^\#$ denotes a singleton type τ , or returns \emptyset , otherwise.

C. *References*: $\llbracket r \rrbracket_r^\# : \mathbb{E}^\# \rightarrow \mathbb{T}^\#$

- Variable Lookups:

$$\llbracket x \rrbracket_r^\#(\sigma^\#) = \sigma^\#(x)$$

- Field Lookups:

$$\llbracket r[e] \rrbracket_r^\#(\sigma^\#) = \{\tau[v] \mid \tau \in \llbracket r \rrbracket_r^\#(\sigma^\#) \wedge v \in \llbracket e \rrbracket_e^\#(\sigma^\#)\}$$

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

D. *Expressions*: $\llbracket e \rrbracket_e^\# : \mathbb{E}^\# \rightarrow \mathbb{T}^\#$

- Completion Records:

$$\begin{aligned} &\llbracket \text{Completion } \{\dots, \text{Type} : e_0, \text{Value} : e_1, \dots\} \rrbracket_e^\#(\sigma^\#) \\ &= \begin{cases} \{\text{normal}(\tau) \mid \tau \in \llbracket e_1 \rrbracket_e^\#(\sigma^\#)\} & \text{if } \llbracket e_0 \rrbracket_e^\# = c_{\text{normal}} \\ \{\text{abrupt}\} & \text{otherwise} \end{cases} \end{aligned}$$

- Records:

$$\llbracket t \{ \dots \} \rrbracket_e^\#(\sigma^\#) = \{t\}$$

- Lists:

$$\begin{aligned} \llbracket [] \rrbracket_e^\#(\sigma^\#) &= [] \\ \llbracket [e_1, \dots, e_n] \rrbracket_e^\#(\sigma^\#) &= \{\tau \mid \tau \in \bigsqcup_{1 \leq i \leq n} \llbracket e_i \rrbracket_e^\#(\sigma^\#)\} \end{aligned}$$

- Type Checks:

$$\llbracket e : \tau \rrbracket_e^\#(\sigma^\#) = \{\tau' <: \tau \mid \tau' \in \llbracket e \rrbracket_e^\#(\sigma^\#)\}$$

- Existence Checks:

$$\llbracket r? \rrbracket_e^\#(\sigma^\#) = \{\tau \neq ? \mid \tau \in \llbracket e \rrbracket_e^\#(\sigma^\#)\}$$

- Binary Operations:

$$\llbracket e_0 \oplus e_1 \rrbracket_e^\#(\sigma^\#) = \{\tau_0 \oplus \tau_1 \mid \tau_0 \in \tau_0^\# \wedge \tau_1 \in \tau_1^\#\}$$

where

$$\tau_0^\# = \llbracket e_0 \rrbracket_e^\#(\sigma^\#) \wedge \tau_1^\# = \llbracket e_1 \rrbracket_e^\#(\sigma^\#)$$

- Unary Operations:

$$\llbracket \ominus e \rrbracket_e^\#(\sigma^\#) = \{\ominus \tau \mid \tau \in \llbracket e \rrbracket_e^\#(\sigma^\#)\}$$

- References:

$$\llbracket r \rrbracket_e^\#(\sigma^\#) = \llbracket r \rrbracket_r^\#(\sigma^\#) \setminus \{?\}$$

- Constants:

$$\llbracket c \rrbracket_e^\#(\sigma^\#) = c$$

- Primitives:

$$\llbracket p \rrbracket_e^\#(\sigma^\#) = \begin{cases} \text{num} & \text{if } p = n \\ \text{bigint} & \text{if } p = j \\ \text{symbol} & \text{if } p = @s \\ p & \text{otherwise} \end{cases}$$