Advanced Occurrence Typing for ECMA-262

ANONYMOUS AUTHOR(S)

This is the technical report of the FSE 2026 paper for "Towards Precise Type Analysis on JavaScript Language Specifications via Occurrence Typing". It first defines the base type analysis on $\mathsf{IR}_{\mathsf{ES}}$, the imperative-style untyped intermediate representation for the JavaScript language specification called ECMA-262. Then, it extends the base type analysis with an advanced occurrence typing technique.

1 Base Type Analysis

This section defines the base type analysis for JavaScript language specification using an abstract interpretation [1, 2] framework with *syntactic type refinement* used in prior work [3].

1.1 $\mathsf{IR}_{\mathsf{ES}}$: Untyped Intermediate Representation for ECMA-262

An $\mathsf{IR}_{\mathsf{ES}}$ program is a mechanized specification and its instructions are uniquely labeled with $l \in \mathcal{L}$.

Syntax. Figure 1 shows the syntax of $\mathsf{IR}_{\mathsf{ES}}$. A program P consists of three mappings: 1) $\mathsf{func} : \mathcal{L} \rightharpoonup \mathcal{F}$, 2) $\mathsf{inst} : \mathcal{L} \to \mathcal{I}$, and 3) $\mathsf{next} : \mathcal{L} \rightharpoonup \mathcal{L}$ that map labels to their enclosing functions, instructions, and next labels, respectively, where $\rightharpoonup$ denotes a partial function. An instruction $i \in \mathcal{I}$ is either a sequence, an assignment, a record allocation, a field update, a branch, an assertion, a function call, or a return instruction. An expression $e \in \mathcal{E}$ is either a literal, a variable, a unary/binary operation, a function, or a dynamic type check expression. We use the $\overline{X_k}$ notation to denote a sequence of elements $X_1, \ldots, X_n$ with an index k or $\overline{X}$ when the index is not important.

Types. A dynamic type check expression "$e : \tau$" checks if the expression e is an instance of the type τ. A type $\tau \in \mathcal{T}$ is either literal (e.g., `42`, `#t`, `"abc"`), a primitive (**B** for booleans, **N** for numbers, **S** for strings), a function, a record, a union, an intersection, a difference, or a top/bottom type. We define a type concretization γ_τ and following auxiliary operations.

- **Type Concretization**: $\gamma_\tau : \mathcal{T} \to \mathcal{P}(\mathbb{H} \times \mathbb{V})$

$$\begin{array}{lll} \gamma_\tau(p) = \mathbb{H} \times \{p\} & \gamma_\tau(\Lambda) = \mathbb{H} \times \{\Lambda\} & \gamma_\tau(\tau_1 \setminus \tau_2) = \gamma_\tau(\tau_1) \setminus \gamma_\tau(\tau_2) \\ \gamma_\tau(\mathbf{B}) = \mathbb{H} \times \mathbb{B} & \gamma_\tau(\tau_1 \sqcap \tau_2) = \gamma_\tau(\tau_1) \cap \gamma_\tau(\tau_2) & \gamma_\tau(\top) = \mathbb{H} \times \mathbb{V} \\ \gamma_\tau(\mathbf{N}) = \mathbb{H} \times \mathbb{N} & \gamma_\tau(\tau_1 \sqcup \tau_2) = \gamma_\tau(\tau_1) \cup \gamma_\tau(\tau_2) & \gamma_\tau(\bot) = \varnothing \end{array}$$

$$\gamma_\tau(\{\overline{\mathsf{f}_k : \tau_k}\}) = \{(h, a) \mid a \in \mathbb{A} \wedge h \in \mathbb{H} \wedge h(a) = \{\overline{\mathsf{f}'_i : v_i}\} \wedge \forall k. \exists i. (\mathsf{f}_k = \mathsf{f}'_i \wedge (h, v_i) \in \gamma_\tau(\tau_k))\}$$

- **Subtype Relation**: $<: \subseteq \mathcal{T} \times \mathcal{T}$

$$\tau_1 <: \tau_2 \iff \gamma_\tau(\tau_1) \subseteq \gamma_\tau(\tau_2)$$

- **Field Access**: $\tau.\mathsf{f} \in \mathcal{T}$

$$p.\mathsf{f} = \bot \quad \mathbf{B}.\mathsf{f} = \bot \quad \mathbf{N}.\mathsf{f} = \bot \qquad \{\overline{\mathsf{f}_k : \tau_k}\}.\mathsf{f} = \begin{cases} \tau_k & \text{if } \mathsf{f} = \mathsf{f}_k \\ \bot & \text{otherwise} \end{cases}$$

$$\begin{array}{lll} (\tau_1 \sqcup \tau_2).\mathsf{f} = \tau_1.\mathsf{f} \sqcup \tau_2.\mathsf{f} & \Lambda.\mathsf{f} = \bot \\ (\tau_1 \sqcap \tau_2).\mathsf{f} = \tau_1.\mathsf{f} \sqcap \tau_2.\mathsf{f} & \top.\mathsf{f} = \top \\ (\tau_1 \setminus \tau_2).\mathsf{f} = \tau_1.\mathsf{f} \setminus \tau_2.\mathsf{f} & \bot.\mathsf{f} = \bot \end{array}$$

- **Field Update**: $\tau[\mathsf{f} : \tau] \in \mathcal{T}$

$$\begin{array}{lll} p[\mathsf{f} : \tau] = \bot & (\tau_1 \sqcup \tau_2)[\mathsf{f} : \tau_f] = \tau_1[\mathsf{f} : \tau_f] \sqcup \tau_2[\mathsf{f} : \tau_f] & \Lambda[\mathsf{f} : \tau] = \bot \\ \mathbf{B}[\mathsf{f} : \tau] = \bot & (\tau_1 \sqcap \tau_2)[\mathsf{f} : \tau_f] = \tau_1[\mathsf{f} : \tau_f] \sqcap \tau_2[\mathsf{f} : \tau_f] & \top[\mathsf{f} : \tau] = \{\mathsf{f} : \tau\} \\ \mathbf{N}[\mathsf{f} : \tau] = \bot & (\tau_1 \setminus \tau_2)[\mathsf{f} : \tau_f] = \tau_1[\mathsf{f} : \tau_f] \setminus \tau_2[\mathsf{f} : \tau_f] & \bot[\mathsf{f} : \tau] = \bot \end{array}$$

2025. ACM XXXX-XXXX/2025/9-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

$$
\begin{array}{lll}
\textbf{Programs} & P = (\text{func}, \text{inst}, \text{next}) & \textbf{Functions} \quad \mathcal{F} \ni \Lambda ::= \lambda(\overline{x}).l \\
\textbf{Instructions} & \mathcal{I} \ni i ::= x = e \mid x = \{\overline{\mathtt{f} : e}\} \mid e.\mathtt{f} = e & \textbf{Labels} \quad \mathcal{L} \ni l \\
 & \quad \mid \mathtt{if}\ (e)\ l\ \mathtt{else}\ l \mid \mathtt{assert}\ e & \textbf{Variables} \quad \mathcal{X} \ni x \\
 & \quad \mid x = e(\overline{e}) \mid \mathtt{return}\ e & \textbf{Fields} \quad \mathbb{F} \ni \mathtt{f} \\
\textbf{Expressions} & \mathcal{E} \ni e ::= p \mid x \mid e.\mathtt{f} \mid \ominus e \mid e \oplus e \mid \Lambda \mid e : \tau & \textbf{Primitives} \quad \mathbb{P} \ni p ::= b \mid n \mid s \\
\textbf{Types} & \mathcal{T} \ni \tau ::= p \mid \mathbf{B} \mid \mathbf{N} \mid \mathbf{S} \mid \Lambda \mid \{\overline{\mathtt{f} : \tau}\} & \textbf{Booleans} \quad \mathbb{B} \ni b ::= \#\mathtt{t} \mid \#\mathtt{f} \\
 & \quad \mid \tau \sqcup \tau \mid \tau \sqcap \tau \mid \tau \backslash \tau \mid \top \mid \bot & \textbf{Numbers} \quad \mathbb{N} \ni n \\
 & & \textbf{Strings} \quad \mathbb{S} \ni s
\end{array}
$$

Fig. 1. Syntax of the *imperative*-style untyped intermediate representation $\mathsf{IR}_{\mathsf{ES}}$ for ECMA-262.

$$
\begin{array}{llll}
\textbf{States} & \sigma \in \Sigma = \mathbb{K} \times \mathcal{L} \times \mathbb{M} & \textbf{Calling Contexts} & \kappa \in \mathbb{K} = \{\epsilon\} \uplus \Sigma \\
\textbf{Memories} & m \in \mathbb{M} = \mathbb{H} \times \mathbb{E} & \textbf{Heaps} & h \in \mathbb{H} = \mathbb{A} \rightharpoonup \mathbb{R} \\
\textbf{Environments} & \rho \in \mathbb{E} = \mathcal{X} \rightharpoonup \mathbb{V} & \textbf{Addresses} & a \in \mathbb{A} \\
\textbf{Values} & v \in \mathbb{V} = \mathbb{P} \uplus \mathbb{A} \uplus \mathcal{F} & \textbf{Records} & r \in \mathbb{R} = \mathbb{F} \rightharpoonup \mathbb{V}
\end{array}
$$

Fig. 2. States of $\mathsf{IR}_{\mathsf{ES}}$ programs.

States. As described in Figure 2, a program state $\sigma \in \Sigma$ is a tuple of a calling context, a label, and a memory. A calling context $\kappa \in \mathbb{K}$ is an empty context ϵ or a state at the call site. A memory $m \in \mathbb{M}$ is a tuple of a heap $h \in \mathbb{H}$ and an environment $\rho \in \mathbb{E}$. A heap $h \in \mathbb{H}$ is a finite map from addresses to records. An address $a \in \mathbb{A}$ is a unique identifier for each record. A record $r \in \mathbb{R}$ is a finite map from fields to values. An environment $\rho \in \mathbb{E}$ is a finite map ($\rightharpoonup$) from variables to values. A value $v \in \mathbb{V}$ is either a primitive, an address, or a function value. Note that variables and fields are *mutable* by assignments and field updates.

Semantics. The concrete semantics of the core language is defined as a state transition system $(\rightsquigarrow, \Sigma, \Sigma_\iota)$ where Σ_ι is a set of initial states and $\rightsquigarrow \subseteq \Sigma \times \Sigma$ is a transition relation between states:

$$\sigma \rightsquigarrow \sigma' \iff [\![i]\!]_i(\sigma) = \sigma' \quad \text{where} \quad \sigma = (l, \_, \_, \_, \_) \wedge i = \mathsf{inst}(l)$$

The *collecting semantics* is defined as a set of reachable states from the initial states and can be computed by a fixed-point computation of a transfer function $F : \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$:

$$[\![P]\!] = \{\sigma \in \Sigma \mid \exists \sigma_\iota \in \Sigma_\iota.\ \sigma_\iota \rightsquigarrow^* \sigma\} = \mathbf{lfp} F = \lim_{n \to \infty} F^n(\varnothing) \quad \text{where} \quad F(S) = \Sigma_\iota \cup \bigcup_{\sigma \in S} \{\sigma' \mid \sigma \rightsquigarrow \sigma'\}$$

The transition-style semantics of instructions $[\![-]\!]_i : \Sigma \to \Sigma$ and expressions $[\![-]\!]_e : \mathbb{M} \to \mathbb{V}$ are defined in Figure 3 and we use the following auxiliary operations:

$$
\begin{array}{llll}
\sigma.\mathsf{caller} = \kappa & \kappa.\mathsf{state} = \sigma & m.\mathsf{heap} & = h \\
\sigma.\mathsf{label} = l & \kappa.\mathsf{caller} = \sigma.\mathsf{caller} & m.\mathsf{env} & = \rho \\
\sigma.\mathsf{mem} = m & \kappa.\mathsf{label} = \sigma.\mathsf{label} & m.\mathsf{update}_x(x, v) & = (h, \rho[x \mapsto v]) \\
\sigma.\mathsf{heap} = m.\mathsf{heap} & \kappa.\mathsf{mem} = \sigma.\mathsf{mem} & m.\mathsf{update}_{\mathtt{f}}(a, \mathtt{f}, v) & = (h[a \mapsto h(a)[\mathtt{f} \mapsto v]], \rho) \\
\sigma.\mathsf{env} = m.\mathsf{env} & \kappa.\mathsf{heap} = \sigma.\mathsf{heap} & m.\mathsf{alloc}(r) & = (a, (h[a \mapsto r], \rho)) \\
\text{where } \sigma = (\kappa, l, m) & \kappa.\mathsf{env} = \sigma.\mathsf{env} & & \text{where } a \text{ is a fresh address in } h \\
 & \kappa.\mathsf{var} = x & \text{where } m = (h, \rho) & \\
 & \text{where } \kappa = (\sigma, x) & &
\end{array}
$$

$$\boxed{[\![-]\!]_i : \Sigma \to \Sigma}$$

$$
\begin{array}{lll}
[\![x = e]\!]_i(\sigma) & = (\kappa, \mathsf{next}(l), m_1) & \text{where } m_1 = m.\mathsf{update}_x(x, [\![e]\!]_e(m)) \\
[\![x = \{\overline{\mathtt{f}_k : e_k}\}]\!]_i(\sigma) & = (\kappa, \mathsf{next}(l), m_2) & \text{where } \begin{cases} (a, m_1) = m.\mathsf{alloc}(\{\overline{\mathtt{f}_k : [\![e_k]\!]_e(m)}\}) \\ m_2 = m_1.\mathsf{update}_x(x, a) \end{cases} \\
[\![e_1.\mathtt{f} = e_2]\!]_i(\sigma) & = (\kappa, \mathsf{next}(l), m_1) & \text{where } m_1 = m.\mathsf{update}_{\mathtt{f}}([\![e_1]\!]_e(m), \mathtt{f}, [\![e_2]\!]_e(m)) \\
[\![\mathtt{if}\ (e)\ l_1\ \mathtt{else}\ l_2]\!]_i(\sigma) & = (\kappa, l', m) & \text{where } l' = \begin{cases} l_1 \text{ if } [\![e]\!]_e(m) = \#\mathtt{t} \\ l_2 \text{ if } [\![e]\!]_e(m) = \#\mathtt{f} \end{cases} \\
[\![\mathtt{assert}(e)]\!]_i(\sigma) & = (\kappa, \mathsf{next}(l), m) & \text{if } \quad [\![e]\!]_e(m) = \#\mathtt{t} \\
[\![\mathtt{call}\ x = e_0(\overline{e_k})]\!]_i(\sigma) & = (\kappa_0, l_0, m_1) & \text{where } \begin{cases} \kappa_0 = \sigma \\ [\![e_0]\!]_e(m) = \lambda(\overline{x_k}).l_0 \\ m_1 = (h, \{\overline{x_k \mapsto [\![e_k]\!]_e(m)}\}) \end{cases} \\
[\![\mathtt{return}\ e]\!]_i(\sigma) & = (\kappa_1, \mathsf{next}(l_1), m_3,) & \text{where } \begin{cases} \kappa = (\kappa_1, l_1, (\_, \rho_1)) \\ \mathsf{inst}(l_1) = \mathtt{call}\ x = \_; \\ m_2 = (h, \rho_1) \\ m_3 = m_2.\mathsf{update}_x(x, [\![e]\!]_e(m)) \end{cases}
\end{array}
$$

$$\boxed{[\![-]\!]_e : \mathbb{M} \to \mathbb{V}}$$

$$
\begin{array}{llll}
[\![p]\!]_e(m) = p & [\![e.\mathtt{f}]\!]_e(m) & = h([\![e]\!]_e(m))(\mathtt{f}) & \\
[\![x]\!]_e(m) = \rho(x) & [\![\ominus\ e]\!]_e(m) & = \ominus\ [\![e]\!]_e(m) & [\![e : \tau]\!]_e(m) = \begin{cases} \#\mathtt{t} \text{ if } (h, [\![e]\!]_e(m)) \in \gamma_\tau(\tau) \\ \#\mathtt{f} \text{ otherwise} \end{cases} \\
[\![\Lambda]\!]_e(m) = \Lambda & [\![e_1 \oplus e_2]\!]_e(m) & = [\![e_1]\!]_e(m) \oplus [\![e_2]\!]_e(m) &
\end{array}
$$

Fig. 3. Semantics of instructions and expressions of $\mathsf{IR}_{\mathsf{ES}}$ where $\sigma = (\kappa, l, m)$ and $m = (h, \rho)$.

Abstract States	$\widehat{\sigma} \in \widehat{\Sigma}$	$=$	$\widehat{\mathbb{K}} \times \Xi$
Abstract Calling Contexts	$\widehat{\kappa} \in \widehat{\mathbb{K}}$	$=$	$\mathcal{F} \to \mathcal{P}(\mathcal{L})$
Flow-Sensitive Typing Results	$\xi \in \Xi$	$=$	$\mathcal{L} \to \mathbb{\Gamma}$
Typing Results	$\Gamma \in \mathbb{\Gamma}$	$=$	$\widehat{\mathbb{E}} \times \Delta$
Abstract Environments	$\widehat{\rho} \in \widehat{\mathbb{E}}$	$=$	$\mathcal{X} \to \widehat{\mathbb{V}}$
Abstract Values	$\widehat{v} \in \widehat{\mathbb{V}}$	$=$	$\mathcal{T}$
Effects	$\delta \in \Delta$	$=$	$\mathbb{F} \to \mathcal{T}$

Fig. 4. Abstract domains for each component of the type analysis.

1.2 Type Analysis for $\mathsf{IR}_{\mathsf{ES}}$

Abstract Domains. Figure 4 presents the abstract domains used in the base type analysis. An abstract state $\widehat{\sigma} \in \widehat{\Sigma}$ consists of a calling context and a flow-sensitive typing result. The calling context $\widehat{\kappa} \in \widehat{\mathbb{K}}$ maps functions to call-site labels, while the typing result $\xi \in \Xi$ maps labels to typing results. Each typing result $\Gamma \in \mathbb{\Gamma}$ pairs an abstract environment and an effect: the environment $\widehat{\rho} \in \widehat{\mathbb{E}}$ maps variables to their types, and the effect $\delta \in \Delta$ maps fields to types to capture possible updates in the current context. Effects ensure soundness by propagating update information across function calls. Standard operations (concretization γ, partial order $\sqsubseteq$, bottom $\bot$, join $\sqcup$, and meet $\sqcap$) are defined point-wise over these domains. Their definitions are defined in Figure 5.

$$\boxed{\begin{array}{llll}\gamma_\sigma : \widehat{\Sigma} \to \mathcal{P}(\Sigma) & \gamma_\kappa : \widehat{\mathbb{K}} \to \mathcal{P}(\Sigma) & \gamma_\xi : \Xi \to \mathcal{P}(\Sigma) & \gamma_\Gamma : \mathbb{\Gamma} \to \mathcal{P}(\Sigma) \\ \gamma_\rho : \widehat{\mathbb{E}} \to \mathcal{P}(\mathbb{M}) & \gamma_v : \widehat{\mathbb{V}} \to \mathcal{P}(\mathbb{H} \times \mathbb{V}) & \gamma_\delta : \Delta \to \mathcal{P}(\Sigma) & \end{array}}$$

$$\begin{array}{l}\gamma_\sigma(\widehat{\sigma}) = \{\sigma \mid \widehat{\sigma} = (\widehat{\kappa}, \xi) \wedge \sigma \in \gamma_\kappa(\widehat{\kappa}) \wedge \sigma \in \gamma_\xi(\xi)\} \\ \gamma_\kappa(\widehat{\kappa}) = \{\sigma \mid \kappa = \sigma.\mathsf{caller} \wedge (\kappa = \epsilon \vee (\kappa.\mathsf{state} \in \gamma_\kappa(\widehat{\kappa}) \wedge (\kappa.\mathsf{label}, \kappa.\mathsf{var}) \in \widehat{\kappa} \circ \mathsf{func}(l)))\} \\ \gamma_\xi(\xi) = \{\sigma \mid \sigma \in \gamma_\Gamma \circ \xi(\sigma.\mathsf{label}) \wedge \kappa = \sigma.\mathsf{caller} \wedge (\kappa = \epsilon \vee \kappa.\mathsf{state} \in \gamma_\xi(\xi))\} \\ \gamma_\Gamma(\Gamma) = \{\sigma \mid \Gamma = (\widehat{\rho}, \delta) \wedge \sigma.\mathsf{mem} \in \gamma_\rho(\widehat{\rho}) \wedge \sigma \in \gamma_\delta(\delta)\} \\ \gamma_\rho(\widehat{\rho}) = \{m \mid m = (h, \rho) \wedge \forall (x \mapsto v) \in \rho.\ (h, v) \in \gamma_v \circ \widehat{\rho}(x)\} \\ \gamma_v(\tau) = \gamma_\tau(\tau) \\ \gamma_\delta(\delta) = \{\sigma \mid \kappa = \sigma.\mathsf{caller} \wedge (\kappa = \epsilon \vee (\kappa.\mathsf{heap}, \sigma.\mathsf{heap}) \in_\delta \delta)\} \\ \text{where } (h, h') \in_\delta \delta \iff \forall a \mapsto r \in h.\ \forall \mathtt{f} \mapsto v \in r.\ v' = h'(a)(\mathtt{f}) \wedge (v' \neq v \Rightarrow (h', v') \in \gamma_\tau \circ \delta(\mathtt{f}))\end{array}$$

$$\boxed{\widehat{d} \sqsubseteq \widehat{d}}$$

$$\widehat{d}_1 \sqsubseteq \widehat{d}_2 \iff \gamma_d(\widehat{d}_1) \subseteq \gamma_d(\widehat{d}_2)$$

$$\boxed{\bot_d}$$

$$\begin{array}{llll}\bot_\sigma = (\bot_\kappa, \bot_\xi) & \bot_\xi = \{l \mapsto \bot_\Gamma \mid l \in \mathcal{L}\} & \bot_\rho = \{x \mapsto \bot_v \mid x \in \mathcal{X}\} & \bot_v = \bot \\ \bot_\kappa = \{\Lambda \mapsto \varnothing \mid \Lambda \in \mathcal{F}\} & \bot_\Gamma = (\bot_\rho, \bot_\delta) & \bot_\delta = \{\mathtt{f} \mapsto \bot \mid \mathtt{f} \in \mathbb{F}\} & \end{array}$$

$$\boxed{\widehat{d} \sqcup \widehat{d} = \widehat{d}}$$

$$\Gamma_1 \sqcup \Gamma_2 = (\widehat{\rho}_1 \sqcup \widehat{\rho}_2, \delta_1 \sqcup \delta_2) \text{ where } \Gamma_1 = (\widehat{\rho}_1, \delta_1) \text{ and } \Gamma_2 = (\widehat{\rho}_2, \delta_2)$$

$$\begin{array}{ll}\widehat{\sigma}_1 \sqcup \widehat{\sigma}_2 = (\widehat{\kappa}_1 \sqcup \widehat{\kappa}_2, \xi_1 \sqcup \xi_2) & \widehat{\rho}_1 \sqcup \widehat{\rho}_2 = \{x \mapsto \widehat{\rho}_1(x) \sqcup \widehat{\rho}_2(x) \mid x \in \mathcal{X}\} \\ \widehat{\kappa}_1 \sqcup \widehat{\kappa}_2 = \{\Lambda \mapsto \widehat{\kappa}_1(\Lambda) \cup \widehat{\kappa}_2(\Lambda) \mid \Lambda \in \mathcal{F}\} & \delta_1 \sqcup \delta_2 = \{\mathtt{f} \mapsto \delta_1(\mathtt{f}) \sqcup \delta_2(\mathtt{f}) \mid \mathtt{f} \in \mathbb{F}\} \\ \xi_1 \sqcup \xi_2 = \{l \mapsto \xi_1(l) \sqcup \xi_2(l) \mid l \in \mathcal{L}\} & \widehat{v}_1 \sqcup \widehat{v}_2 = \widehat{v}_1 \sqcup \widehat{v}_2\end{array}$$

$$\boxed{\widehat{d} \sqcap \widehat{d} = \widehat{d}}$$

$$\Gamma_1 \sqcap \Gamma_2 = (\widehat{\rho}_1 \sqcap \widehat{\rho}_2, \delta_1 \sqcap \delta_2) \text{ where } \Gamma_1 = (\widehat{\rho}_1, \delta_1) \text{ and } \Gamma_2 = (\widehat{\rho}_2, \delta_2)$$

$$\begin{array}{ll}\widehat{\sigma}_1 \sqcap \widehat{\sigma}_2 = (\widehat{\kappa}_1 \sqcap \widehat{\kappa}_2, \xi_1 \sqcap \xi_2) & \widehat{\rho}_1 \sqcap \widehat{\rho}_2 = \{x \mapsto \widehat{\rho}_1(x) \sqcap \widehat{\rho}_2(x) \mid x \in \mathcal{X}\} \\ \widehat{\kappa}_1 \sqcap \widehat{\kappa}_2 = \{\Lambda \mapsto \widehat{\kappa}_1(\Lambda) \cup \widehat{\kappa}_2(\Lambda) \mid \Lambda \in \mathcal{F}\} & \delta_1 \sqcap \delta_2 = \{\mathtt{f} \mapsto \delta_1(\mathtt{f}) \sqcap \delta_2(\mathtt{f}) \mid \mathtt{f} \in \mathbb{F}\} \\ \xi_1 \sqcap \xi_2 = \{l \mapsto \xi_1(l) \sqcap \xi_2(l) \mid l \in \mathcal{L}\} & \widehat{v}_1 \sqcap \widehat{v}_2 = \widehat{v}_1 \sqcap \widehat{v}_2\end{array}$$

Fig. 5. Concretization (γ), partial order ($\sqsubseteq$), bottom ($\bot$), join ($\sqcup$), and meet ($\sqcap$) on abstract domains.

Abstract Semantics. The abstract semantics $\widehat{[\![P]\!]}$ of a program is defined as a fixed-point computation of the abstract transfer function $\widehat{F} : \widehat{\Sigma} \to \widehat{\Sigma}$ from an initial abstract state $\widehat{\sigma}_\iota$:

$$\widehat{[\![P]\!]} = \mathbf{lfp}\widehat{F} = \lim_{n \to \infty} (\widehat{F})^n(\bot_\sigma) \qquad \text{where} \qquad \widehat{F}(\widehat{\sigma}) = \widehat{\sigma}_\iota \sqcup \left(\bigsqcup_{l \in \mathcal{L}} \widehat{[\![\mathsf{inst}(l)]\!]}_i(l, \widehat{\sigma}) \right)$$

where $\widehat{\sigma}_\iota$ satisfies the constraints $\Sigma_\iota \subseteq \gamma_\sigma(\widehat{\sigma}_\iota)$ and $\widehat{[\![-]\!]}_i$ and $\widehat{[\![-]\!]}_e$ are the abstract semantics of instructions and expressions, respectively. Figure 6 shows the abstract semantics of branch and assertion instructions that perform type refinements; the rest are in the supplementary material. For abstract semantics, we introduce the auxiliary operations for typing results $\Gamma = (\widehat{\rho}, \delta) \in \mathbb{\Gamma}$:

$$
\begin{array}{lll}
\widehat{[\![x = e]\!]}_i(l,\widehat{\sigma}) & = (\varnothing, \{\mathsf{next}(l) \mapsto \Gamma_1\}) & \text{where } \Gamma_1 = \Gamma.\widehat{\mathsf{update}}_x(x, \widehat{[\![e]\!]}_e(\Gamma)) \\
\widehat{[\![x = \{\mathtt{f}_k : e_k\}]\!]}_i(l,\widehat{\sigma}) & = (\varnothing, \{\mathsf{next}(l) \mapsto \Gamma_1\}) & \text{where } \Gamma_1 = \Gamma.\widehat{\mathsf{update}}_x(x, \{\mathtt{f}_k : \widehat{[\![e_k]\!]}_e(\Gamma)\}) \\
\widehat{[\![e_1.\mathtt{f} = e_2]\!]}_i(l,\widehat{\sigma}) & = (\varnothing, \{\mathsf{next}(l) \mapsto \Gamma_1\}) & \text{where } \Gamma_1 = \Gamma.\widehat{\mathsf{update}}_\mathtt{f}(\widehat{[\![e_1]\!]}_e(\Gamma), \mathtt{f}, \widehat{[\![e_2]\!]}_e(\Gamma)) \\
\widehat{[\![\mathtt{if}\ (e)\ l_1\ \mathtt{else}\ l_2]\!]}_i(l,\widehat{\sigma}) & = (\varnothing, \{l_t \mapsto \Gamma_t, l_f \mapsto \Gamma_f\}) & \text{where } \begin{cases} \Gamma_t = \Gamma.\widehat{\mathsf{refine}}(e, \#\mathtt{t}) \\ \Gamma_f = \Gamma.\widehat{\mathsf{refine}}(e, \#\mathtt{f}) \end{cases} \\
\widehat{[\![\mathtt{assert}(e)]\!]}_i(l,\widehat{\sigma}) & = (\varnothing, \{\mathsf{next}(l) \mapsto \Gamma_1\}) & \text{where } \Gamma_1 = \Gamma.\widehat{\mathsf{refine}}(e, \#\mathtt{t}) \\
\widehat{[\![\mathtt{call}\ x = e_0(\overline{e_k})]\!]}_i(l,\widehat{\sigma}) & = (\widehat{\kappa}_1, \{\overline{l_j \mapsto \Gamma_j}\}) & \text{where } \begin{cases} \widehat{v}_0 = \widehat{[\![e_0]\!]}_e(\Gamma) \\ \{\Lambda_j\} = \{\widehat{\lambda(\overline{x_{j,k}}).l_j} = \gamma_v(\widehat{v}_0) \cap \mathcal{F}\} \\ \Gamma_j = (\{x_{j,k} \mapsto \widehat{[\![e_k]\!]}_e(\Gamma)\}, \bot_\delta) \\ \widehat{\kappa}_1 = \{\Lambda_j \mapsto \{l\}\} \end{cases} \\
\widehat{[\![\mathtt{return}\ e]\!]}_i(l,\widehat{\sigma}) & = (\varnothing, \{\overline{\mathsf{next}(l_k) \mapsto \Gamma'_k}\}) & \text{where } \begin{cases} \{\overline{l_k}\} = \widehat{\kappa} \circ \mathsf{func}(l) \\ \mathsf{inst}(l_k) = \mathtt{call}\ x_k = \_; \\ \Gamma_k = \xi(l_k).\widehat{\mathsf{apply}}(\delta) \\ \Gamma'_k = \Gamma_k.\widehat{\mathsf{update}}_x(x_k, \widehat{[\![e]\!]}_e(\Gamma)) \end{cases}
\end{array}
$$

$$
\widehat{[\![-]\!]}_e : \Gamma \to \widehat{\mathbb{V}}
$$

$$
\widehat{[\![p]\!]}_e(\Gamma) = p \quad \widehat{[\![x]\!]}_e(\Gamma) = \widehat{\rho}(x) \quad \widehat{[\![\ominus\, e]\!]}_e(\Gamma) = \ominus \widehat{[\![e]\!]}_e(\Gamma) \quad \widehat{[\![e : \tau]\!]}_e(\Gamma) = \mathbf{B}
$$

$$
\widehat{[\![\Lambda]\!]}_e(\Gamma) = \Lambda \quad \widehat{[\![e.\mathtt{f}]\!]}_e(\Gamma) = \widehat{[\![e]\!]}_e(\Gamma).\mathtt{f} \quad \widehat{[\![e_1 \oplus e_2]\!]}_e(\Gamma) = \widehat{[\![e_1]\!]}_e(\Gamma) \oplus \widehat{[\![e_2]\!]}_e(\Gamma)
$$

Fig. 6. Abstract semantics of instructions and expressions of $\mathsf{IR}_{\mathsf{ES}}$ where $\widehat{\sigma} = (\widehat{\kappa}, \xi)$ and $\xi(l) = \Gamma = (\widehat{\rho}, \delta)$.

- $\Gamma.\widehat{\mathsf{update}}_x : (\mathcal{X} \times \widehat{\mathbb{V}}) \to \mathbb{\Gamma}$ updates a variable in a typing result with a new abstract value.

$$\Gamma.\widehat{\mathsf{update}}_x(x, \widehat{v}) = (\widehat{\rho}[x \mapsto \widehat{v}], \delta)$$

- $\Gamma.\widehat{\mathsf{update}}_\mathtt{f} : (\widehat{\mathbb{V}} \times \mathbb{F} \times \widehat{\mathbb{V}}) \to \mathbb{\Gamma}$ updates a field of an abstract value in a typing result.

$$\Gamma.\widehat{\mathsf{update}}_\mathtt{f}(\_, \mathtt{f}, \widehat{v}_2) = (\{x \mapsto (\widehat{v} \sqcup \widehat{v}[\mathtt{f} : \widehat{v}_2]) \mid (x \mapsto \widehat{v}) \in \widehat{\rho}\}, \delta[\mathtt{f} : \widehat{v}_2])$$

- $\Gamma.\widehat{\mathsf{apply}} : \Delta \to \mathbb{\Gamma}$ applies an effect (may-update information of fields) to a typing result.

$$\Gamma.\widehat{\mathsf{apply}}(\delta) = \bigsqcup\{\Gamma.\widehat{\mathsf{update}}_\mathtt{f}(\top_{\widehat{v}}, \mathtt{f}, \tau) \mid (\mathtt{f} : \tau) \in \delta\}$$

- $\Gamma.\widehat{\mathsf{refine}} : (\mathcal{E} \times \mathbb{B}) \to \mathbb{\Gamma}$ performs a *syntactic type refinement* on a typing result in branch/assertion instructions with conditional expressions as highlighted in Figure 6.

$$
\Gamma.\widehat{\mathsf{refine}}(e_1\ \&\&\ e_2, b) = \begin{cases} \Gamma_1 \sqcap \Gamma_2 \text{ if } b \\ \Gamma_1 \sqcup \Gamma_2 \text{ if } \neg b \end{cases} \qquad \Gamma.\widehat{\mathsf{refine}}(!\,e, b) = \Gamma.\widehat{\mathsf{refine}}(e, \neg b)
$$

$$
\Gamma.\widehat{\mathsf{refine}}(e_1\ ||\ e_2, b) = \begin{cases} \Gamma_1 \sqcup \Gamma_2 \text{ if } b \\ \Gamma_1 \sqcap \Gamma_2 \text{ if } \neg b \end{cases} \qquad \Gamma.\widehat{\mathsf{refine}}(x : \tau, b) = \begin{cases} (\widehat{\rho}[x \mapsto \widehat{\rho}(x) \sqcap \tau], \delta) \text{ if } b \\ (\widehat{\rho}[x \mapsto \widehat{\rho}(x) \setminus \tau], \delta) \quad \text{if } \neg b \end{cases}
$$

where $\Gamma_1 = \Gamma.\widehat{\mathsf{refine}}(e_1, b)$ and $\Gamma_2 = \Gamma.\widehat{\mathsf{refine}}(e_2, b)$.

2 Type Analysis with Advanced Occurrence Typing

This section extends the base type analysis on ECMA-262 with our advanced occurrence typing to increase the analysis precision. First, we extend abstract domains type with type guards (§2.1) and define the abstract semantics (§2.2). Then, we define provenances of type refinements as their explanation and extend the abstract domain to track provenances (§2.3).

Extended Abstract State	$\widehat{\sigma} \in \widehat{\Sigma}$	$=$	$\widehat{\mathbb{K}} \times \Xi$
Extended Abstract State	$\widehat{\sigma} \in \widehat{\Sigma}$	$=$	$\widehat{\mathbb{K}} \times \Xi$
Abstract Calling Contexts	$\widehat{\kappa} \in \widehat{\mathbb{K}}$	$=$	$\mathcal{F} \to \mathcal{P}(\mathcal{L})$
Flow-Sensitive Typing Results	$\xi \in \Xi$	$=$	$\mathcal{L} \to \mathbb{\Gamma}$
Typing Results	$\Gamma \in \mathbb{\Gamma}$	$=$	$\widehat{\mathbb{E}} \times \Psi \times \Delta$
Abstract Environments	$\widehat{\rho} \in \widehat{\mathbb{E}}$	$=$	$\mathcal{X} \to \widehat{\mathbb{V}}$
Abstract Values	$\widehat{v} \in \widehat{\mathbb{V}}$	$=$	$\mathcal{T}^{\omega} \times \mathbb{G}$
Effects	$\delta \in \Delta$	$=$	$\mathbb{F} \to \mathcal{T}$
Type Propositions	$\psi \in \Psi$	$::=$	$\top_{\psi} \mid \bot_{\psi} \mid x : \tau \mid \omega <: \tau \mid \psi \wedge \psi \mid \ldots$
Symbols (Type Variables)	$\omega \in \Omega$	$=$	$\{\omega^{\mathtt{0}}, \omega^{1}, \ldots\}$
Symbolic Types	$\tau^{\omega} \in \mathcal{T}^{\omega}$	$::=$	$\tau \mid \omega \mid \tau^{\omega}.\mathtt{f} \mid \{\overline{\mathtt{f} : \tau^{\omega}}\}$
Type Guards	$G \in \mathbb{G}$	$=$	$\mathcal{T}^{D} \to \Psi$
Demanded Types	$\tau^{D} \in \mathcal{T}^{D}$	$\subset$	$\mathcal{T}$

Fig. 7. Abstract domains for extended type analysis with occurrence typing.

2.1 Extended Abstract Domain

Figure 7 shows the extended part of abstract domains. We first introduce three new components with symbols: 1) *type propositions* to represent refinements on types of variables and symbols, 2) *symbolic types* for type aliasing, and 3) *type guards* to support type propositions for user-defined demanded types. Using these components, we extend the abstract domain to support the occurrence typing with user-defined demanded types as highlighted in green. A typing result $\Gamma \in \mathbb{\Gamma}$ is augmented with a type proposition $\psi \in \Psi$ that always holds at the current point in the specification. An abstract value $\widehat{v} \in \widehat{\mathbb{V}}$ is not just a type but a symbolic type $\tau^{\omega} \in \mathcal{T}^{\omega}$ with a corresponding type guard $G \in \mathbb{G}$.

Type Propositions. A *type proposition* $\psi \in \Psi$ is a proposition on types of variables and symbols. A *symbol* (or a *type variable*) $\omega \in \Omega$ represents an argument type flowed from callers; $\omega^{\mathtt{0}}, \omega^{1}, \ldots$ denote the types of the first, second, and so on arguments. The top ($\top_{\psi}$) and bottom ($\bot_{\psi}$) type propositions always hold and never hold, respectively. For example, a type proposition $\psi = (x : \mathbf{N}) \wedge (\omega^{\mathtt{0}} <: \mathbf{S})$ denotes that the variable x is the number type $\mathbf{N}$ and the type of the first argument is a subtype of the string type $\mathbf{S}$. We use only conjunctions of type propositions on variables or symbols for simplicity and efficiency of analysis in this paper, but other operations (e.g., disjunctions) or other kinds of type propositions can be freely added. While type propositions are basically inferred from a dynamic type check expression $(e : \tau)$, they can be inferred for other expressions with extended rules. The analysis utilizes type propositions to refine types of variables and symbols in branches.

Symbolic Types. Since the equality relation is crucial for type refinement, we represent aliases between types using symbolic types. A *symbolic type* $\tau^{\omega} \in \mathcal{T}^{\omega}$ is either a concrete τ, a symbol ω, a symbolic field access $\tau^{\omega}.\mathtt{f}$, or a symbolic record type $\{\overline{\mathtt{f} : \tau^{\omega}}\}$. Figure 8a shows the operations on symbolic types. The upper type bound $\lceil \tau^{\omega} \rceil_{\psi}$ denotes the upper bound of τ^{ω} under the type proposition ψ. The lattice operations ($\sqsubseteq$, $\sqcup$, and $\sqcap$) on symbolic types are defined based on it.

The refinement operation ($\downarrow$) on symbolic types is the key operation used in the type refinement in the extended type analysis. Note that $\lceil \tau^{\omega} \rceil_{\psi}$ is the upper type bound not the actual type of the symbolic type τ^{ω} because they can be instantiated with any concrete types satisfying the type propositions ψ. It means that the actual type of a symbolic type τ^{ω} is not fixed until the type propositions are applied to it. Symbolic types that share the same symbols can be refined together using the type propositions on the symbols. For example, variables x and y in the abstract

$$\frac{}{\lceil \tau \rceil_\psi = \tau} \qquad \frac{\psi \Rightarrow (\omega <: \tau)}{\lceil \omega \rceil_\psi = \tau} \qquad \frac{\lceil \tau^\omega \rceil_\psi = \tau}{\lceil \tau^\omega.\mathtt{f} \rceil_\psi = \tau.\mathtt{f}} \qquad \frac{\overline{\lceil \tau_k^\omega \rceil_\psi = \tau_k}}{\lceil \{\overline{s_k : \tau_k^\omega}\} \rceil_\psi = \{\overline{s_k : \tau_k}\}}$$

(a) Upper type bound $\lceil \tau^\omega \rceil_\psi$ operations on symbolic types τ^ω with a type proposition ψ.

$$\tau_1^\omega|_{\psi_1} \sqsubseteq \tau_2^\omega|_{\psi_2} \Leftrightarrow \begin{cases} \lceil \tau_1^\omega \rceil_{\psi_1} <: \tau_2 & \text{if } \tau_2^\omega = \tau_2 \\ \tau_1^\omega = \tau_2^\omega & \text{otherwise} \end{cases} \qquad \tau_1^\omega|_{\psi_1} \sqcup \tau_2^\omega|_{\psi_2} = \begin{cases} \tau_1^\omega & \text{if } \tau_1^\omega = \tau_2^\omega \\ \lceil \tau_1^\omega \rceil_{\psi_1} \sqcup \lceil \tau_2^\omega \rceil_{\psi_2} & \text{otherwise} \end{cases}$$

$$\tau_1^\omega|_{\psi_1} \sqcap \tau_2^\omega|_{\psi_2} = \begin{cases} \tau_1^\omega & \text{if } \tau_1^\omega \sqsubseteq \tau_2^\omega \\ \tau_2^\omega & \text{if } \tau_2^\omega \sqsubseteq \tau_1^\omega \\ \lceil \tau_1^\omega \rceil_{\psi_1} \sqcap \lceil \tau_2^\omega \rceil_{\psi_2} & \text{otherwise} \end{cases} \qquad \tau_1^\omega|_{\psi_1} \downarrow \tau_2 = \begin{cases} \tau_1^\omega & \text{if } \tau_1^\omega \sqsubseteq \tau_2^\omega \\ \lceil \tau_1^\omega \rceil_{\psi_1} \sqcap \tau_2^\omega & \text{otherwise} \end{cases}$$

(b) Partial order ($\sqsubseteq$), join ($\sqcup$), meet ($\sqcap$), and refinement ($\downarrow$) for symbolic types τ^ω with type propositions.

Fig. 8. Operations on symbolic types $\tau^\omega \in \mathcal{T}^\omega$.

Type Guard Lookup $\quad G(\tau) = \bigsqcap \{\psi \mid (\tau^D \Rightarrow \psi) \in G \wedge (\tau <: \tau^D)\}$

Type Guard Update $\quad G[\tau \Rightarrow \psi] = \lambda \tau^D. \begin{cases} G(\tau^D) \sqcap \psi & \text{if } \tau <: \tau^D \\ G(\tau^D) & \text{otherwise} \end{cases}$

Type Guard Refinement $\quad G \downarrow \tau = \lambda \tau^D. \begin{cases} \bot_\psi & \text{if } \tau \sqcap \tau^D = \bot_\tau \\ G(\tau^D) & \text{otherwise} \end{cases}$

Type Guard Field Lookup $\quad G.\mathtt{f} = \top_G[\overline{\tau_k^D.\mathtt{f} \Rightarrow \psi_k}]$ where $G = \{\overline{\tau_k^D \Rightarrow \psi_k}\}$

Type Guard Field Update $\quad G[\mathtt{f} : \tau] = \top_G[\overline{\tau_k^D[\mathtt{f} : \tau] \Rightarrow \psi_k[\mathtt{f} : \tau]}]$ where $G = \{\overline{\tau_k^D \Rightarrow \psi_k}\}$

Record Type Guard $\quad \{\overline{\mathtt{f}_j : G_j}\} = \top_G[\overline{\{\mathtt{f}_j : \tau_k^D\} \Rightarrow \psi_{j,k}}]$ where $G_j = \{\overline{\tau_j^D \Rightarrow \psi_{j,k}}\}$

Fig. 9. Operations on type guards $G \in \mathbb{G}$.

environment $\widehat{\rho} \in \widehat{\mathbb{E}}$ are refined together when the type propositions ψ_1 and ψ_2 are applied:

$$\widehat{\rho} = [\ x \mapsto \omega,\ y \mapsto \{\mathtt{f} : \omega\}\ ] \quad \text{means} \quad \begin{cases} [\ x \mapsto \mathbf{N} \sqcup \mathbf{S},\quad y \mapsto \{\mathtt{f} : \mathbf{N} \sqcup \mathbf{S}\}\ ] & \text{if } \psi_1 = \omega <: \mathbf{N} \sqcup \mathbf{S} \\ [\ x \mapsto \mathbf{N},\quad y \mapsto \{\mathtt{f} : \mathbf{N}\}\ ] & \text{if } \psi_2 = \omega <: \mathbf{N} \end{cases}$$

It also supports parametric polymorphism by instantiating symbols with concrete argument types.

Type Guards. A *type guard* $G \in \mathbb{G}$ is a mapping from user-defined demanded types $\mathcal{T}^D$ to type propositions Ψ. A single mapping $\tau^D \Rightarrow \psi$ states that if a value is a demanded type τ^D, then the type proposition ψ always holds. Type guards allow the chained application of multiple propositions. For example, a type proposition $\psi_1 = (x : \mathbf{N})$ is applied to an abstract environment $\widehat{\rho}$:

$$\widehat{\rho} = [\ x \mapsto (\mathbf{B} \sqcup \mathbf{N})\ [\mathbf{N} \Rightarrow \overline{(y : \mathbf{S})}^{\psi_2}],\quad y \mapsto \mathbf{B} \sqcup \mathbf{S}\ ] \quad \xrightarrow{\text{applying } \overline{(x : \mathbf{N})}^{\psi_1}} \quad [\ x \mapsto \mathbf{N}\ [\ldots],\quad y \mapsto \mathbf{S}\ ]$$

Then, ψ_1 makes the type of x be refined from $\mathbf{B} \sqcup \mathbf{N}$ to $\mathbf{N}$, causing ψ_2 in the guard to hold as well. Thus, the type of y is also refined from $\mathbf{B} \sqcup \mathbf{S}$ to $\mathbf{S}$.

The lattice operations ($\sqsubseteq$, $\sqcup$, and $\sqcap$) on type guards are defined in a point-wise manner, and other operations on type guards are defined in Figure 9. The operations $G(\tau)$ and $G[\tau \Rightarrow \psi]$ lookup and update propositions associated with demanded types that are supertypes of the given type τ. The guard refinement $G \downarrow \tau$ refines the type guard to remove impossible propositions for a given type τ. The field lookup $G.\mathtt{f}$, field update $G[\mathtt{f} : \tau]$, and record $\{\overline{\mathtt{f} : G}\}$ operations are defined via update operations and produce a new type guard by extracting propositions associated with the demanded

types after field lookup, field update, and record allocation, respectively. The notation $\psi[\mathtt{f} : \tau]$ means that it updates the type of field `f` in every record type within the type proposition ψ to τ while dropping other fields' types to $\bot_\tau$. For example, consider the following type guard G with five demanded types $\mathcal{T}^D = \{\#\mathtt{t}, \#\mathtt{f}, \mathbf{B}, \mathbf{N}, \{\mathtt{f} : \mathbf{N}\}\}$:

$$G = [\ \#\mathtt{t} \Rightarrow \psi_{\#\mathtt{t}},\quad \#\mathtt{f} \Rightarrow \psi_{\#\mathtt{f}},\quad \mathbf{B} \Rightarrow \psi_{\mathbf{B}},\quad \mathbf{N} \Rightarrow \psi_{\mathbf{N}}\quad \{\mathtt{f} : \mathbf{N}\} \Rightarrow \psi_{\mathtt{f}:\mathbf{N}}\ ]$$

Then, the result of each operation on the type guard G is as follows:

$$\begin{array}{lllll}
G(\#\mathtt{t}) & = \psi_{\#\mathtt{t}} \sqcap \psi_{\mathbf{B}} & & & \\
G[\#\mathtt{t} \Rightarrow \psi] & = [\ \#\mathtt{t} \Rightarrow \psi_{\#\mathtt{t}} \sqcap \psi, & \#\mathtt{f} \Rightarrow \psi_{\#\mathtt{f}}, & \mathbf{B} \Rightarrow \psi_{\mathbf{B}} \sqcap \psi, & \mathbf{N} \Rightarrow \psi_{\mathbf{N}} \qquad \{\mathtt{f} : \mathbf{N}\} \Rightarrow \psi_{\mathtt{f}:\mathbf{N}}\ ] \\
G \downarrow \mathbf{B} & = [\ \#\mathtt{t} \Rightarrow \psi_{\#\mathtt{t}}, & \#\mathtt{f} \Rightarrow \psi_{\#\mathtt{f}}, & \mathbf{B} \Rightarrow \psi_{\mathbf{B}}, & \mathbf{N} \Rightarrow \bot_\psi \qquad \{\mathtt{f} : \mathbf{N}\} \Rightarrow \bot_\psi\ ] \\
G.\mathtt{f} & = [\ \#\mathtt{t} \Rightarrow \bot_\psi, & \#\mathtt{f} \Rightarrow \bot_\psi, & \mathbf{B} \Rightarrow \bot_\psi, & \mathbf{N} \Rightarrow \psi_{\mathtt{f}:\mathbf{N}} \qquad \{\mathtt{f} : \mathbf{N}\} \Rightarrow \bot_\psi\ ] \\
G[\mathtt{f} : \mathbf{N}] & = [\ \#\mathtt{t} \Rightarrow \bot_\psi, & \#\mathtt{f} \Rightarrow \bot_\psi, & \mathbf{B} \Rightarrow \bot_\psi, & \mathbf{N} \Rightarrow \psi_{\mathtt{f}:\mathbf{N}}[\mathtt{f} : \mathbf{N}], \{\mathtt{f} : \mathbf{N}\} \Rightarrow \bot_\psi\ ] \\
\{\mathtt{f} : G\} & = [\ \#\mathtt{t} \Rightarrow \bot_\psi, & \#\mathtt{f} \Rightarrow \bot_\psi, & \mathbf{B} \Rightarrow \bot_\psi, & \mathbf{N} \Rightarrow \bot_\psi, \qquad \{\mathtt{f} : \mathbf{N}\} \Rightarrow \psi_{\mathbf{N}}\ ]
\end{array}$$

When a type guard G is bound to a specific symbolic type τ^ω in an abstract value (i.e., $\widehat{v} = \tau^\omega[G]$)), the refinement operation is used for removing infeasible propositions (i.e., $G \downarrow \lceil \tau^\omega \rceil_\psi$).

In addition, we can reduce the size of the type guard by skipping meaningless propositions that always hold or never hold according to the context. If the current typing result is $\Gamma = (\widehat{\rho}, \psi, \delta)$ and a skipped proposition ψ' is bound to a demanded type τ^D for the symbolic type τ^ω in an abstract value, then ψ' is $\bot_\psi$ if $\lceil \tau^\omega \rceil_\psi \sqcap \tau^D = \bot_\psi$ or ψ otherwise. For example, consider the following abstract value $\widehat{v}$ with three demanded types $\mathcal{T}^D = \{\mathbf{N}, \#\mathtt{t}, \#\mathtt{f}\}$:

$$\widehat{v} = \mathbf{B}\ [\ \#\mathtt{f} \Rightarrow \psi_{\#\mathtt{f}}\ ] \quad \text{means} \quad \mathbf{B}\ [\ \mathbf{N} \Rightarrow \bot_\psi \quad \#\mathtt{t} \Rightarrow \psi,\quad \#\mathtt{f} \Rightarrow \psi_{\#\mathtt{f}}\ ]$$

Then, the skipped propositions associated with demanded types $\mathbf{N}$ and $\#\mathtt{t}$ are $\bot_\psi$ and ψ, respectively, because $\mathbf{B} \sqcap \mathbf{N} = \bot_\psi$ but $\mathbf{B} \sqcap \#\mathtt{t} = \#\mathtt{t} \neq \bot_\psi$. If an abstract value $\widehat{v} = \tau^\omega[G]$ having a top type guard $G = \top_G$, we simply write τ^ω instead of $\tau^\omega[\top_G]$. The lattice operations on other abstract domains are defined in the same way as the original abstract domain in §1.2 or extended in a point-wise manner. For example, $\{\overline{\mathtt{f}_k : \widehat{v}_k}\} = \{\overline{\mathtt{f}_k : \tau_k^\omega}\}\ [\{\overline{\mathtt{f}_k : G_k}\}]$ where $\widehat{v}_k = \tau_k^\omega[G_k]$.

2.2 Extended Abstract Semantics

This section describes the extended abstract semantics as defined in Figure 10 and highlights the modified or newly added parts in green. In the extended abstract semantics, type propositions are 1) **inferred** when analyzing expressions or producing abstract values, 2) **propagated** into other points even across functions, 3) **weakened** when variables or fields are mutated, and 4) used to **refine** types of variables and symbols in branch or assertion instructions.

Type Proposition Inference. Type propositions are inferred using the $\widehat{\mathsf{infer}}_G : (\mathbb{T} \times \mathcal{E}) \to \mathbb{G}$ operation. It basically generates a type guard G for a dynamic type check $e : \tau$ with two propositions: $\psi_{\#\mathtt{t}}$ (when e is a subtype of τ) for $\#\mathtt{t}$ and $\psi_{\#\mathtt{f}}$ (when disjoint from τ) for $\#\mathtt{f}$. Additional rules can extend type proposition inference for other expressions. Type propositions are also inferred from produced abstract values via the operation $\widehat{v}.\widehat{\mathsf{bind}} : \Psi \to \mathbb{G}$, which checks whether the upper type of the produced abstract value $\widehat{v}$ is a subtype of any demanded type τ^D and binds the proposition ψ in the current typing result to the demanded type τ^D in the type guard. The $\widehat{\mathsf{bind}}$ operation is invoked whenever an abstract value is produced (e.g., results of expressions, allocated records, returned values, etc.) in the abstract semantics.

Type Proposition Propagation. The inferred type propositions are propagated into other points even across functions. When calling a function, analysis introduces indexed symbols ω^{k} to represent type variables for the actual argument types. The inferred type propositions for the arguments

$$\boxed{\widehat{[\![-]\!]}_i : (\mathcal{L} \times \widehat{\Sigma}) \to \widehat{\Sigma}}$$

$$\begin{array}{lll}
\widehat{[\![x = e]\!]}_i(l, \widehat{\sigma}) & = (\varnothing, \{\mathsf{next}(l) \mapsto \Gamma_1\}) & \text{where } \Gamma_1 = \Gamma.\widehat{\mathsf{update}}_x(x, \widehat{[\![e]\!]}_e(\Gamma)) \\
\widehat{[\![x = \{\mathtt{f}_k : e_k\}]\!]}_i(l, \widehat{\sigma}) & = (\varnothing, \{\mathsf{next}(l) \mapsto \Gamma_1\}) & \text{where } \Gamma_1 = \Gamma.\widehat{\mathsf{update}}_x(x, \{\mathtt{f}_k : \widehat{[\![e_k]\!]}_e(\Gamma)\}).\,\widehat{\mathsf{bind}}(\psi) \\
\widehat{[\![e_1.\mathtt{f} = e_2]\!]}_i(l, \widehat{\sigma}) & = (\varnothing, \{\mathsf{next}(l) \mapsto \Gamma_1\}) & \text{where } \Gamma_1 = \Gamma.\widehat{\mathsf{update}}_{\mathtt{f}}(\widehat{[\![e_1]\!]}_e(\Gamma), \mathtt{f}, \widehat{[\![e_2]\!]}_e(\Gamma)) \\
\widehat{[\![\mathtt{if}\ (e)\ l_1\ \mathtt{else}\ l_2]\!]}_i(l, \widehat{\sigma}) & = (\varnothing, \{l_t \mapsto \Gamma_t, l_f \mapsto \Gamma_f\}) & \text{where } \begin{cases} \Gamma_t = \Gamma.\widehat{\mathsf{refine}}(e, \#\mathtt{t}) \\ \Gamma_f = \Gamma.\widehat{\mathsf{refine}}(e, \#\mathtt{f}) \end{cases} \\
\widehat{[\![\mathtt{assert}(e)]\!]}_i(l, \widehat{\sigma}) & = (\varnothing, \{\mathsf{next}(l) \mapsto \Gamma_1\}) & \text{where } \Gamma_1 = \Gamma.\widehat{\mathsf{refine}}(e, \#\mathtt{t}) \\
\widehat{[\![\mathtt{call}\ x = e_0(\overline{e_k})]\!]}_i(l, \widehat{\sigma}) & = (\widehat{\kappa}_1, \{\overline{l_j \mapsto \Gamma_j}\}) & \text{where } \begin{cases} \widehat{v}_0 = \widehat{[\![e_0]\!]}_e(\Gamma) \\ (\tau_k^{\omega}, _) = \widehat{[\![e_k]\!]}_e(\Gamma) \\ \psi' = \bigwedge\{\omega^{\mathsf{k}} <: \lceil \tau_k^{\omega} \rceil_{\psi}\} \\ \{\Lambda_j\} = \{\overline{\lambda(\overline{x_{j,k}}).l_j}\} = \gamma_v(\widehat{v}_0) \cap \mathcal{F} \\ \Gamma_j = (\{\overline{x_{j,k} \mapsto \omega^{\mathsf{k}}}\}, \psi', \bot_{\delta}) \\ \widehat{\kappa}_1 = \{\overline{\Lambda_j \mapsto (l, x)}\} \end{cases} \\
\widehat{[\![\mathtt{return}\ e]\!]}_i(l, \widehat{\sigma}) & = (\varnothing, \{\overline{\mathsf{next}(l_k) \mapsto \Gamma_k'}\}) & \text{where } \begin{cases} \{\overline{(l_k, x_k)}\} = \widehat{\kappa} \circ \mathsf{func}(l) \\ \mathsf{inst}(l_k) = \mathtt{call}\ x_k = _(\overline{e_{k,j}}) \\ \Gamma_k = \xi(l_k).\widehat{\mathsf{apply}}(\delta) \\ \widehat{v} = \widehat{[\![e]\!]}_e(\Gamma) \\ \widehat{v}_k = \widehat{v}.\,\mathsf{instantiate}(\Gamma_k, \overline{e_{k,j}}).\,\widehat{\mathsf{bind}}(\psi) \\ \Gamma_k' = \Gamma_k.\widehat{\mathsf{update}}_x(x_k, \widehat{v}_k) \end{cases}
\end{array}$$

$$\boxed{\widehat{[\![-]\!]}_e : \mathbb{\Gamma} \to \widehat{\mathbb{V}}}$$

$$\widehat{[\![e]\!]}_e(\Gamma) = \ldots [\ G\ ].\,\widehat{\mathsf{bind}}(\psi) \ \text{ where } \ G = \widehat{\mathsf{infer}}_G(\Gamma, e)$$

Fig. 10. Extended abstract semantics where $\widehat{\sigma} = (\widehat{\kappa}, \xi)$ and $\xi(l) = \Gamma = (\widehat{\rho}, \psi, \delta)$.

inside the function are propagated into the call-site stored in the type guard of the returned abstract values. Then, the type propositions are instantiated with the argument expressions and their actual abstract values using the operation $\widehat{v}.\mathsf{instantiate} : (\mathbb{\Gamma} \times \mathcal{E}) \to \widehat{\mathbb{V}}$. For example, consider an argument expression e_{arg}, its abstract value $\widehat{v}_{\text{arg}}$, and the returned abstract value $\widehat{v}_{\text{ret}}$ after calling a function:

$$e_{\text{arg}} = x.\mathtt{f} \qquad \widehat{v}_{\text{arg}} = \{\mathtt{g} : \mathbf{N} \sqcup \mathbf{S}\}[\{\mathtt{g} : \mathbf{N}\} \Rightarrow \psi] \qquad \widehat{v}_{\text{ret}} = \omega^{\emptyset}.\mathtt{g}[\mathbf{S} \Rightarrow (\omega^{\emptyset} <: \mathbf{B})]$$

Let $\tau^{\omega}[G]$ be the instantiated returned abstract value with above information. Then,

$$\tau^{\omega} = \mathbf{N} \sqcup \mathbf{S} \qquad G = G_{\text{arg}} \sqcap G_{\text{ret}} \qquad G_{\text{arg}} = [\mathbf{N} \Rightarrow \psi] \qquad G_{\text{ret}} = [\mathbf{S} \Rightarrow x : \{\mathtt{f} : \mathbf{B}\}]$$

Therefore, $\tau^{\omega}[G] = \mathbf{N} \sqcup \mathbf{S}[\mathbf{N} \Rightarrow \psi, \mathbf{S} \Rightarrow x : \{\mathtt{f} : \mathbf{B}\}]$. The interesting point is that both type guards 1) G_{arg} existing in the argument abstract value and 2) G_{ret} inferred in another function are propagated and merged into the type guard G of the instantiated returned abstract value.

Weakening Type Propositions. Since our target language allows mutation of variables and fields, we need to weaken the type propositions in the type guard when they are mutated for sound type analysis. As shown in Figure 13, type propositions can be weakened through the update operations for variables $\Gamma.\widehat{\mathsf{update}}_x$ and fields $\Gamma.\widehat{\mathsf{update}}_{\mathtt{f}}$. When a variable x is mutated with a new abstract value $\widehat{v}$, all propositions associated with x are invalid and should be removed from the type

$$\widehat{\mathsf{infer}}_G(\Gamma, e : \tau) = \top_G[\#\mathtt{t} \Rightarrow \psi_{\#\mathtt{t}}][\#\mathtt{f} \Rightarrow \psi_{\#\mathtt{f}}] \quad \text{where} \begin{cases} (\tau^\omega, \_) = \widehat{[\![e]\!]}_e(\Gamma) \\ \psi_{\#\mathtt{t}} = \widehat{\mathsf{infer}}_\psi(e, \tau^\omega)(\lceil \tau^\omega \rceil_\psi \sqcap \tau) \\ \psi_{\#\mathtt{f}} = \widehat{\mathsf{infer}}_\psi(e, \tau^\omega)(\lceil \tau^\omega \rceil_\psi \setminus \tau) \end{cases}$$

$$\widehat{\mathsf{infer}}_G(\Gamma, e) = \top_G \quad \text{otherwise}$$

(a) The type guard inference operation $\widehat{\mathsf{infer}}_G : (\mathbb{\Gamma} \times \mathcal{E}) \to \mathbb{G}$ where $\Gamma = (\widehat{\rho}, \psi, \delta)$.

$$\widehat{\mathsf{infer}}_\psi(e, \tau^\omega)(\tau) = \begin{cases} \psi & \text{if } \tau^\omega \in \mathcal{T} \wedge (e <:? \tau \Downarrow \psi) \\ \psi & \text{if } \tau^\omega \notin \mathcal{T} \wedge (\tau^\omega <:? \tau \Downarrow \psi) \\ \top_\psi & \text{otherwise} \end{cases}$$

where $e <:? \tau \Downarrow \psi$ or $\tau^\omega <:? \tau \Downarrow \psi$ is defined as follows:

$$\frac{}{x <:? \tau \Downarrow x : \tau} \quad \frac{e <:? \{\mathtt{f} : \tau\} \Downarrow \psi}{e.\mathtt{f} <:? \tau \Downarrow \psi} \quad \frac{}{\omega <:? \tau \Downarrow \omega <: \tau} \quad \frac{\tau^\omega <:? \{\mathtt{f} : \tau\} \Downarrow \psi}{\tau^\omega.\mathtt{f} <:? \tau \Downarrow \psi} \quad \frac{\overline{\tau_k^\omega <:? \tau.\mathtt{f}_k \Downarrow \psi_k}}{\{\mathtt{f}_k : \tau_k^\omega\} <:? \tau \Downarrow \bigwedge\{\overline{\psi_k}\}}$$

(b) The type proposition inference operation $\widehat{\mathsf{infer}}_\psi : (\mathcal{E} \times \mathcal{T}^\omega) \to \mathcal{T} \to \Psi$.

$$\tau^\omega[G].\widehat{\mathsf{bind}}(\psi) = \tau^\omega[G'] \text{ where } G' = G[\lceil \tau^\omega \rceil_\psi \Rightarrow \psi]$$

(c) The binding operation $\widehat{v}.\widehat{\mathsf{bind}} : \Psi \to \mathbb{G}$ that binds type propositions to demanded types in type guards.

Fig. 11. Operations related to **type proposition inference**.

$$\tau^\omega[G].\mathsf{instantiate}(\Gamma, \overline{e_k}) = \tau_0^\omega[G_0 \sqcap G'] \text{ where } \begin{cases} \widehat{v}_k = \tau_k^\omega[G_k] = \widehat{[\![e_k]\!]}_e(\Gamma) \\ \tau_0^\omega = \tau^\omega[\overline{\omega^{\mathsf{k}} \mapsto \tau_k^\omega}] \\ G_0 = \tau^\omega[\overline{\omega^{\mathsf{k}} \mapsto G_k}] \\ G' = \lambda\tau^D.G(\tau^D).\mathsf{instantiate}([\overline{(e_k, \tau_k^\omega)}]) \end{cases}$$

$$\text{and } \psi.\mathsf{instantiate}([\overline{(e_k, \tau_k^\omega)}]) = \begin{cases} \bot_\psi & \text{if } \psi = \bot_\psi \\ \psi_0.\mathsf{instantiate}(\overline{(e_k, \tau_k^\omega)}) \wedge \psi_1.\mathsf{instantiate}(\overline{(e_k, \tau_k^\omega)}) & \text{if } \psi = \psi_0 \wedge \psi_1 \\ \widehat{\mathsf{infer}}_\psi(e_k, \tau_k^\omega)(\tau) & \text{if } \psi = \omega^{\mathsf{k}} <: \tau \\ \top_\psi & \text{otherwise} \end{cases}$$

Fig. 12. Instantiation $\widehat{v}.\mathsf{instantiate} : (\mathbb{\Gamma} \times \mathcal{E}) \to \widehat{\mathbb{V}}$ of abstract values with argument expressions for inter-procedural **type proposition propagation**.

guard and typing result. The $\Gamma.\mathsf{weaken}_x(x)$ operation invalidates the propositions associated with x in the typing result Γ while $\Gamma.\mathsf{weaken}_\mathtt{f}(\mathtt{f} : \tau)$ invalidates only the specific field $\mathtt{f}$ in all record types in Γ via weak update with the new type τ. For example, the type of the field $\mathtt{f}$ is weakened from $\mathbf{N}$ to $\mathbf{N} \sqcup \mathbf{B}$ with the new type $\mathbf{B}$ in the following example:

$$\omega <: \{\mathtt{f} : \mathbf{N}, \quad \mathtt{g} : \mathbf{S}\} \xrightarrow{\Gamma.\mathsf{weaken}_\mathtt{f}(\mathtt{f} : \mathbf{B})} \omega <: \{\mathtt{f} : (\mathbf{N} \sqcup \mathbf{B}), \quad \mathtt{g} : \mathbf{S}\}$$

While this formalization uses uniform abstract addresses for all record types – potentially introducing imprecision – we use partitions of abstract addresses to avoid such imprecision in our implementation.

Type Refinement. Unlike syntactic type refinement in the original type analysis, the type refinement utilizes the inferred type propositions in the type guard as shown in Figure 14. If a proposition $x : \tau$ for a variable x is applicable, the abstract value of x is refined as $\widehat{\rho}(x)|_\psi \downarrow \tau$ using the refinement operation ($\downarrow$) for symbolic types and for type guards. In addition, it enables chained

$$\Gamma.\widehat{\mathsf{update}}_x(x,\widehat{v}) = (\widehat{\rho}[x\mapsto\widehat{v}],\psi,\delta).\ \mathsf{weaken}_x(x) \text{ where } \begin{cases} \widehat{\rho}.\mathsf{weaken}(x) &= \{y\mapsto\widehat{v}.\mathsf{weaken}(x) \mid (y\mapsto\widehat{v})\in\widehat{\rho}\} \\ \widehat{v}.\mathsf{weaken}(x) &= \tau^\omega[G.\mathsf{weaken}(x)]\wedge\tau^\omega[G]=\widehat{v} \\ G.\mathsf{weaken}(x) &= \{\tau^D\mapsto\psi.\mathsf{weaken}(x)\mid\tau^D\in\mathcal{T}^D\} \\ \psi.\mathsf{weaken}(x) &= \begin{cases}\bot_\psi \text{ if } \psi = x:\tau \\ \psi \quad \text{otherwise}\end{cases} \end{cases}$$

(a) Modified variable update operation $\Gamma.\widehat{\mathsf{update}}_x : (\mathcal{X}\times\widehat{\mathbb{V}})\to\mathbb{\Gamma}$.

$$\Gamma.\widehat{\mathsf{update}}_{\mathsf{f}}(\widehat{v}_1,\mathsf{f},\widehat{v}_2) = (\{x\mapsto(\widehat{v}\sqcup\widehat{v}[\mathsf{f}:\widehat{v}_2])\mid(x\mapsto\widehat{v})\in\widehat{\rho}\},\delta[\mathsf{f}:\widehat{v}_2]) = \Gamma.\ \mathsf{weaken}_{\mathsf{f}}(\mathsf{f}:\tau)$$

(b) Modified field update operation $\Gamma.\widehat{\mathsf{update}}_{\mathsf{f}} : (\widehat{\mathbb{V}}\times\mathbb{F}\times\widehat{\mathbb{V}})\to\mathbb{\Gamma}$ where $\widehat{v}_2 = \tau_2^\omega[\_]$, and $\tau_2 = \lceil\tau_2^\omega\rceil_\psi$.

Fig. 13. Modified variable/field update operations for **weakening type propositions** where $\Gamma = (\widehat{\rho},\psi,\delta)$.

$$\Gamma.\widehat{\mathsf{refine}}(e,b) = \Gamma.\ \widehat{\mathsf{refine}(\psi)} \text{ where } \_[G] = \widehat{[\![e]\!]}_e(\Gamma) \text{ and } \psi = G[b]$$

$$\text{and } \Gamma.\widehat{\mathsf{refine}}(\psi') = \begin{cases} (\widehat{\rho}[x\mapsto\widehat{v}],\psi\wedge\psi',\delta).\widehat{\mathsf{refine}}(G[\tau]) & \text{if } (\psi' = x:\tau)\wedge\widehat{v} = (\widehat{\rho}(x)|_\psi\downarrow\tau) \\ \Gamma.\widehat{\mathsf{refine}}(\psi_0)\sqcap\Gamma.\widehat{\mathsf{refine}}(\psi_1) & \text{if } \psi' = (\psi_0\wedge\psi_1) \\ (\widehat{\rho},\psi\wedge\psi',\delta) & \text{otherwise} \end{cases}$$

Fig. 14. Modified **type refinement** $\Gamma.\widehat{\mathsf{refine}} : \mathcal{E}\times\mathbb{B}\to\mathbb{\Gamma}$ with $\Gamma.\widehat{\mathsf{refine}} : \Psi\to\mathbb{\Gamma}$ where $\Gamma = (\widehat{\rho},\psi,\delta)$.

$$\tau_{\zeta_1}\sqsubseteq\tau_{\zeta_2} \Leftrightarrow \text{if } \tau_1\neq\tau_2 \text{ then } \tau_1\sqsubseteq\tau_2$$
$$\text{else } \begin{cases} \zeta_1.\mathsf{child}\sqsubseteq\zeta_2.\mathsf{child} & \text{if } \zeta_1=\zeta_2=\mathsf{Root}(l,\_)\mid\mathsf{Call}(l,\_)\mid\mathsf{Join}(l,\_)\mid\mathsf{Meet}(l,\_) \\ \mathtt{\#t} & \text{if } \zeta_1=\zeta_2=\mathsf{Leaf}(l) \\ \mathtt{\#f} & \text{otherwise} \end{cases}$$

(a) Partial order $\sqsubseteq : \mathcal{T}_{\mathbb{Z}}\times\mathcal{T}_{\mathbb{Z}}\to\mathbb{B}$

$$\tau_{\zeta_1}\sqcup\tau_{\zeta_2} = \mathsf{canonical}\left(\begin{cases}\tau_{\zeta_1} & \text{if } \tau_{\zeta_2}\sqsubseteq\tau_{\zeta_1} \\ \tau_{\zeta_2} & \text{if } \tau_{\zeta_1}\sqsubseteq\tau_{\zeta_2} \\ (\tau_1\sqcup\tau_2)_{(\zeta_1\sqcup\zeta_2)} & \text{otherwise}\end{cases}\right)$$

$$\text{where } \zeta_1|_{\tau_1}\sqcup\zeta_2|_{\tau_2} = \begin{cases}\zeta_1 & \text{if } \zeta_1=\zeta_2 \\ \mathsf{Join}(s_1\sqcup s_2) & \text{if } \zeta_1=\mathsf{Join}(s_1)\wedge\zeta_2=\mathsf{Join}(s_2) \\ \mathsf{Join}(s_1\sqcup\{\tau_{\zeta_2}\}) & \text{if } \zeta_1=\mathsf{Join}(s_1)\wedge\zeta_2=\mathsf{Call}\mid\mathsf{Leaf} \\ \mathsf{Join}(s_2\sqcup\{\tau_{\zeta_1}\}) & \text{if } \zeta_2=\mathsf{Join}(s_2)\wedge\zeta_1=\mathsf{Call}\mid\mathsf{Leaf} \\ \mathsf{Join}(\{\tau_{\zeta_1},\tau_{\zeta_2}\}) & \text{otherwise}\end{cases}$$

(b) Join operation $\sqcup : \mathcal{T}_{\mathbb{Z}}\times\mathcal{T}_{\mathbb{Z}}\to\mathcal{T}_{\mathbb{Z}}$

$$\mathsf{canonical}(\mathsf{Join}(\{\overline{\tau_\zeta}\})) = \begin{cases}\mathsf{canonical}(\mathsf{Join}(\{\overline{\tau_\zeta}\}\setminus\{\tau_{\zeta_1}\})) \\ \quad \text{if } \exists\tau_{\zeta_2}\in\{\overline{\tau_\zeta}\}\setminus\{\tau_{\zeta_1}\} \text{ s.t. } (\tau_1<:\tau_2\wedge\tau_1\neq\tau_2) \\ \mathsf{Join}(\{\overline{\mathsf{canonical}(\tau_\zeta)}\}) \quad \text{otherwise}\end{cases}$$
$$\mathsf{canonical}(\tau_\zeta) = \tau_\zeta \quad \text{for } \zeta\neq\mathsf{Join}(\_)\wedge\zeta\neq\mathsf{Meet}(\_)$$

(c) Auxiliary operation canonical : $\mathcal{T}_{\mathbb{Z}}\to\mathcal{T}_{\mathbb{Z}}$

Fig. 15. Domain operations for types with provenance.

application of the type propositions in the type guard as explained in §2.1 by recursively invoking the $\widehat{\mathsf{refine}}$ operation if the result of type refinement satisfies another demanded type $\tau^D\in\mathcal{T}^D$.

2.3 Provenance Tracking

This section introduces the provenance, which augments type propositions to add an explanation of why the type proposition can ensure that refinement is legal.

2.3.1 Definition. A *provenance* is bound with an *explanation* that shows why the refinement can happen; at the same time, it is an abstraction of the propagation path of propositions as a citizen of the abstract domain. It gives the simplest possible call paths set for each variable and each refinement, that affected the type guard used at the point.

$$\begin{array}{lrcl} \textbf{Provenance} & \zeta \in \mathbb{Z} & = & \mathsf{Nodes} \\ \textbf{Type With Provenance} & \tau_\zeta \in \mathcal{T}_\mathbb{Z} & = & \mathcal{T} \times \mathbb{Z} \\ \textbf{Type Propositions} & \psi \in \Psi & = & (\mathcal{X} \uplus \Omega) \to \mathcal{T}_\mathbb{Z} \end{array}$$

$$\mathsf{Nodes} \ni \zeta = \mathsf{Root}(l, \tau_\zeta) \mid \mathsf{Call}(l, \tau_\zeta) \mid \mathsf{Join}(l, \{\overline{\tau_\zeta}\}) \mid \mathsf{Meet}(l, \{\overline{\tau_\zeta}\}) \mid \mathsf{Leaf}(l) \mid \bot \mid \top$$

The type with provenance is the same as the type in the section before, but augmented with the provenance tree. For example, consider a type guard with single element $\{\tau^D \mapsto (x \mapsto \tau_\zeta)\}$. Then, this guard means that the type of x can refined to τ when the associated abstract value evaluates to τ^D or below, and the provenance ζ explains how the bound τ has inferred. In detail, the provenance tracks changes and formulate the history as a tree whenever it instantiated in call, or the type has joined with another one. The provenance tree is made of multiple kinds of nodes representing the path in terms of the analysis point, which means a set of locations we want to track the action (change of the type guard). In this paper, we show an example of provenance with label l as analysis points. Root, Call has a single refinement point with a single child, and Join, Meet has a set of children which can be retrieved by $\zeta.\mathsf{child}$. Most of the definitions are trivial, but we note that Root means the refinement point and Leaf means the generation point of the guard.

2.3.2 Domain Operations. Provenances are not simple logs but rather a part of the abstract domain. This means that if a pair of provenances is merged, we should demonstrate that a provenance represents the simplest path that can explain both provenances. We formally define operations between types with provenance in the supplementary material. We define here a operation between types with provenance as Figure 15, with a slight abuse of annotation with $\tau_{\zeta_i} = (\tau_i, \zeta_i)$. Since rules for $\bot$ and $\top$ are trivial, we omit them. Note that it omits unnecessary nodes if a simpler explanation is possible. For example, let the $\zeta_1 = (\mathbf{N}, \mathsf{Call}(l_1))$ and $\zeta_2 = (\mathbf{N} \mid \mathbf{S}, \mathsf{Call}(l_2))$, then $\zeta_1 \sqcup \zeta_2$ ignores the former since the latter is enough to explain the refinement.

References

[1] 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).* doi:10.1145/512950.512973

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

[3] Jihyeok Park, Seungmin An, Wonho Shin, Yusung Sim, and Sukyoung Ryu. 2021. JSTAR: JavaScript Specification Type Analyzer using Refinement. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE).* doi:10.1109/ASE51524.2021.9678781