notJS Concrete and Abstract Semantics

1 The notJS Abstract Syntax

In the figure below, the vector notation represents an ordered sequence where (by abuse of notation) the sequence is of unspecified length n and the subscript i ranges from 1 to n. Expressions e are pure and guaranteed to terminate without exceptions; statements s are impure. The *Num* domain represents 64-bit double precision floating point values conforming to the IEEE754 standard. There is a syntactic restriction that all **jump**s must be to labels that are declared within the same scope as the **jump** statement.

$$n \in \mathit{Num} \qquad b \in \mathit{Bool} \qquad \mathit{str} \in \mathit{String} \qquad x \in \mathit{Variable} \qquad \ell \in \mathit{Label}$$

$$\begin{aligned}
s \in \mathit{Stmt} ::=\ & \vec{s_i} \ \mid\ \textbf{if}\ e\ s_1\ s_2 \ \mid\ \textbf{while}\ e\ s \ \mid\ x := e \ \mid\ x := e_1(e_2, e_3) \ \mid\ x := \textbf{new}\ e_1(e_2) \\
\mid\ & x := \textbf{newfun}\ m\ n \ \mid\ x := \textbf{toobj}\ e \ \mid\ x := \textbf{del}\ e_1.e_2 \ \mid\ e_1.e_2 := e_3 \\
\mid\ & \textbf{throw}\ e \ \mid\ \textbf{try-catch-fin}\ s_1\, x\, s_2\, s_3 \ \mid\ \ell\ s \ \mid\ \textbf{jump}\ \ell\ e \ \mid\ \textbf{for}\ x\, e\, s \\
e \in \mathit{Exp} ::=\ & n \ \mid\ b \ \mid\ \mathit{str} \ \mid\ \textbf{undef} \ \mid\ \textbf{null} \ \mid\ x \ \mid\ e_1 \oplus e_2 \ \mid\ \odot e \\
d \in \mathit{Decl} ::=\ & \textbf{decl}\ \overrightarrow{x_i = e_i}\ \textbf{in}\ s \\
m \in \mathit{Method} ::=\ & (\texttt{self}, \texttt{args}) \Rightarrow d \ \mid\ (\texttt{self}, \texttt{args}) \Rightarrow s \\
\oplus \in \mathit{BinaryOp} ::=\ & + \mid - \mid \times \mid \div \mid \% \mid \ll \mid \gg \mid ⋙ \mid < \mid \leq \mid \& \mid '|' \mid \veebar \\
\mid\ & \textbf{and} \mid \textbf{or} \mid +\!\!+ \mid \prec \mid \preceq \mid \approx \mid \equiv \mid . \mid \textbf{instanceof} \mid \textbf{in} \\
\odot \in \mathit{UnaryOp} ::=\ & - \mid \sim \mid \neg \mid \textbf{typeof} \mid \textbf{tobool} \mid \textbf{isprim} \mid \textbf{tostr} \mid \textbf{tonum}
\end{aligned}$$

2 Concrete Semantics

In this section we describe the following: (1) the concrete semantic domains; (2) the concrete evaluation of expressions; (3) the concrete state transition rules; and (4) the concrete helper functions used by the above descriptions.

2.1 Concrete Semantic Domains

$$
\begin{aligned}
\varsigma \in \mathit{State} &= \mathit{Term} \times \mathit{Env} \times \mathit{Store} \times \mathit{Kont} \\
t \in \mathit{Term} &= \mathit{Decl} + \mathit{Stmt} + \mathit{Value} \\
\rho \in \mathit{Env} &= \mathit{Variable} \to \mathit{Address} \\
\sigma \in \mathit{Store} &= \mathit{Address} \to (\mathit{BValue} + \mathit{Object}) \\
a \in \mathit{Address} &= \mathbb{N} \\
bv \in \mathit{BValue} &= \mathit{Num} + \mathit{Bool} + \mathit{String} + \mathit{Address} + \{\textsf{null}\} + \{\textsf{undef}\} \\
o \in \mathit{Object} &= (\mathit{String} \to \mathit{BValue}) \times (\mathit{String} \to (\mathit{BValue} + \mathit{Class} + \mathit{Closure})) \\
c \in \mathit{Class} &= \{\textsf{function}, \textsf{array}, \textsf{string}, \textsf{boolean}, \textsf{number}, \textsf{date}, \textsf{error}, \textsf{regexp}, \textsf{arguments}, \textsf{object}\} \\
clo \in \mathit{Closure} &= \mathit{Env} \times \mathit{Method} \\
ev \in \mathit{EValue} &= \textsf{exc}\ \mathit{BValue} \\
jv \in \mathit{JValue} &= \textsf{jmp}\ \mathit{Label} \times \mathit{BValue} \\
v \in \mathit{Value} &= \mathit{BValue} + \mathit{EValue} + \mathit{JValue} \\
\kappa \in \mathit{Kont} &= \textsf{haltK}\ + \\
&\quad\ \textsf{seqK}\ \overrightarrow{\mathit{Stmt}_i}\ \mathit{Kont}\ + \\
&\quad\ \textsf{whileK}\ \mathit{Exp}\ \mathit{Stmt}\ \mathit{Kont}\ + \\
&\quad\ \textsf{forK}\ \overrightarrow{\mathit{String}_i}\ \mathit{Variable}\ \mathit{Stmt}\ \mathit{Kont}\ + \\
&\quad\ \textsf{retK}\ \mathit{Variable}\ \mathit{Env}\ \mathit{Kont}\ \{\textsf{ctor}, \textsf{call}\}\ + \\
&\quad\ \textsf{tryK}\ \mathit{Variable}\ \mathit{Stmt}\ \mathit{Stmt}\ \mathit{Kont}\ + \\
&\quad\ \textsf{catchK}\ \mathit{Stmt}\ \mathit{Kont}\ + \\
&\quad\ \textsf{finK}\ \mathit{Value}\ \mathit{Kont}\ + \\
&\quad\ \textsf{lblK}\ \mathit{Label}\ \mathit{Kont}
\end{aligned}
$$

Note: A BValue is a base value; an EValue is an exception value; and a JValue is a jump value. Objects are a pair of maps: the first map contains the regular programmer-visible properties, the second map contains the internal properties used by the interpreter that are invisible to the programmer. We abuse notation by sometimes treating objects $o \in \mathit{Object}$ directly as maps, e.g., $\mathtt{dom}(o)$, $o(str)$, $o[str \mapsto bv]$, $o - str$; implicitly this means to use the regular, programmer-visible map. We use the notation $\pi_i(tup)$ to project out the i^{th} component of tuple tup, where i can be a numeric index or a domain indicated by the corresponding metavariable. The set of classes *Class* represent those classes for which objects of that class can be created dynamically; there are a set of singleton objects in the initial state that are each of their own singleton class, and those singleton classes are not listed here. The **ctor** and **call** used in **retK** continuation inform whether we are returning from a constructor call or a normal method call.

2.1.1 Property Attributes

Some programmer-visible properties have special property attributes $attr \in \{\textsf{nodelete}, \textsf{noenum}, \textsf{noupdate}\}$. Different object classes have different sets of attributes for specific properties. Any property with the **nodelete** attribute will never be deleted by `delete`; any property with the **noenum** attribute will never be returned by `objKeys`; any property with the **noupdate** attribute will never be updated by `updateObj` (see Section 2.4 for definitions of these functions).

We track the class of each object, and a specific class's property attributes implicitly override the behaviors of `delete`, `objKeys`, and `updateObj`. The classes of objects, the special objects they inherit from, their properties and attributes are all detailed in `builtin.pdf`. There does exist an API to modify property attributes whose use would make this information incorrect, though its use is very rare. This API is easy to detect, and to ensure correct behavior we could dynamically modify the above information if this API is used by a program.

2.2 Concrete Expression Evaluation

Note: In Sections 2.2 and 2.3 we use the notation $[\![\cdot]\!]$ to mean $\eta(\cdot, \rho, \sigma)$ when ρ and σ are obvious from the context. We define an evaluator for pure expressions $e \in \mathit{Exp}$:

$\eta : \mathit{Exp} \times \mathit{Env} \times \mathit{Store} \to \mathit{BValue}$

$$\eta(e, \rho, \sigma) =$$

$$\begin{cases} e & \text{if } e \in \mathit{Num} \cup \mathit{String} \cup \mathit{Bool} \cup \{\textsf{null}, \textsf{undef}\} \\ \sigma(\rho(e)) & \text{if } e \in \mathit{Variable} \\ [\![e_1]\!] \oplus [\![e_2]\!] & \text{if } e = e_1 \oplus e_2 \\ \odot[\![e']\!] & \text{if } e = \odot e' \end{cases}$$

2.2.1 Numeric Binary Operators.

The binary operators $\{+, -, \times, \div, \%, \ll, \gg, \ggg, <, \leq, \&, |, \veebar\}$ are defined on Num. $\{+, -, \times, \div, <, \leq\}$ are the standard mathematical operators. $\%$ is remainder. $\ll$ is left-shift. $\gg$ is right-shift with sign extension. $\ggg$ is right-shift with zero extension. $\{\&, |, \veebar\}$ are bitwise `AND`, `OR`, and `XOR`.

2.2.2 String Binary Operators.

The binary operators $\{+\!\!+, \prec, \preceq\}$ are defined on String. $+\!\!+$ is string concatenation. $\prec$ is strict lexicographic comparison. $\preceq$ is reflexive lexicographic comparison.

2.2.3 Boolean Binary Operators

The binary operators $\{\textsf{and}, \textsf{or}\}$ are defined on Bool.

2.2.4 Strict Equality Operator.

The binary operator $\equiv$ is defined on all BValue domains. The $=_n$ is the equality operator on 64-bit double precision floating point values conforming to the IEEE754 standard. In particular, `NaN` $=_n$ `NaN` is **false**.

$$bv_1 \equiv bv_2 = \begin{cases} n_1 =_n n_2 & \text{if } bv_1 = n_1,\ bv_2 = n_2 \\ \textsf{true} & \text{if } bv_1 = b,\ bv_2 = b \\ \textsf{true} & \text{if } bv_1 = \mathit{str},\ bv_2 = \mathit{str} \\ \textsf{true} & \text{if } bv_1 = a,\ bv_2 = a \\ \textsf{true} & \text{if } bv_1 = \textsf{null},\ bv_2 = \textsf{null} \\ \textsf{true} & \text{if } bv_1 = \textsf{undef},\ bv_2 = \textsf{undef} \\ \textsf{false} & \text{otherwise} \end{cases}$$

2.2.5 Non-strict Equality Operator.

The binary operator $\approx$ is defined on all BValue domains. This is slightly different from the `==` operator in JavaScript; in particular, it does not perform the checks in steps 6, 7, 8 and 9 from Section 11.9.3 in the ECMA-262 standard. These checks are inserted by the translator when translating the JavaScript `==` operator.

$$bv_1 \approx bv_2 = \begin{cases} n_1 \equiv n_2 & \text{if } bv_1 = n_1,\ bv_2 = n_2 \\ \textsf{true} & \text{if } bv_1 = b,\ bv_2 = b \\ \textsf{true} & \text{if } bv_1 = str,\ bv_2 = str \\ \textsf{true} & \text{if } bv_1 = a,\ bv_2 = a \\ \textsf{true} & \text{if } bv_1 = \textsf{null},\ bv_2 = \textsf{null} \\ \textsf{true} & \text{if } bv_1 = \textsf{undef},\ bv_2 = \textsf{undef} \\ \textsf{true} & \text{if } bv_1 = \textsf{null},\ bv_2 = \textsf{undef} \\ \textsf{true} & \text{if } bv_1 = \textsf{undef},\ bv_2 = \textsf{null} \\ n \equiv \textsf{tonum}\ str & \text{if } bv_1 = n,\ bv_2 = str \\ \textsf{tonum}\ str \equiv n & \text{if } bv_1 = str,\ bv_2 = n \\ \textsf{false} & \text{otherwise} \end{cases}$$

2.2.6 The Access Operator.

The binary operator . accesses a property of an object. The translator guarantees it is only applied to an address and string.

$a.str = \texttt{lookup}(a, str, \sigma)$

2.2.7 The instanceof Operator.

The binary operator **instanceof** checks if the object referenced by the right-hand operand is an instance of the object referenced by the left-hand operand. The translator guarantees the left-hand side is an address.

$$bv_1\ \textsf{instanceof}\ a = \begin{cases} \textsf{false} & \text{if } bv_1 \neq a' \\ \texttt{instance}(a', a) & \text{otherwise} \end{cases}$$

where $\texttt{instance}(a_1, a_2) =$

$$\begin{cases} \textsf{true} & \text{if } \texttt{getProto}(\sigma(a_1)) = a_2 \\ \textsf{false} & \text{if } \texttt{getProto}(\sigma(a_1)) = \textsf{null} \\ \texttt{instance}(\texttt{getProto}(\sigma(a_1)), a_2) & \text{otherwise} \end{cases}$$

2.2.8 The in Operator.

The binary operator **in** checks if the string on the right-hand side is a property in the object referenced by the left-hand side, either directly or via prototype-based inheritance. The translator guarantees it is only applied to a string and address.

$str\ \textsf{in}\ a = \texttt{find}(str, a)$

where $\texttt{find}(str, a) =$

$$\begin{cases} \textsf{true} & \text{if } str \in \texttt{dom}(\sigma(a)) \\ \textsf{false} & \text{if } str \notin \texttt{dom}(\sigma(a)),\ \texttt{getProto}(\sigma(a)) = \textsf{null} \\ \texttt{find}(str, \texttt{getProto}(\sigma(a))) & \text{otherwise} \end{cases}$$

2.2.9 Numeric Unary Operators.

The unary operators $\{-, \sim\}$ are defined on *Num*. $-$ is negation. $\sim$ is bitwise `NOT`.

2.2.10 Negation Operator.

The unary operator $\neg$ is defined on *Bool*; it is logical negation.

2.2.11 The typeof Operator.

The unary operator **typeof** returns a string representing the type of the given value.

$$\textsf{\textbf{typeof}}\ bv = \begin{cases} \texttt{"number"} & \text{if } bv \in \mathit{Num} \\ \texttt{"boolean"} & \text{if } bv \in \mathit{Bool} \\ \texttt{"string"} & \text{if } bv \in \mathit{String} \\ \texttt{"object"} & \text{if } bv \in \mathit{Address},\ \texttt{getClass}(\sigma(a)) \neq \textsf{\textbf{function}} \\ \texttt{"function"} & \text{if } bv \in \mathit{Address},\ \texttt{getClass}(\sigma(a)) = \textsf{\textbf{function}} \\ \texttt{"object"} & \text{if } bv = \textsf{\textbf{null}} \\ \texttt{"undefined"} & \text{if } bv = \textsf{\textbf{undef}} \end{cases}$$

2.2.12 The tobool Operator.

The unary operator **tobool** converts a value to a boolean.

$$\textsf{\textbf{tobool}}\ bv = \begin{cases} \textsf{\textbf{false}} & \text{if } bv \in \{\textsf{\textbf{null}}, \textsf{\textbf{undef}}, 0, \texttt{NaN}, \textsf{\textbf{false}}, \texttt{""}\} \\ \textsf{\textbf{true}} & \text{otherwise} \end{cases}$$

2.2.13 The isprim Operator.

The unary operator **isprim** determines whether a value is primitive or references an object.

$$\textsf{\textbf{isprim}}\ bv = \begin{cases} \textsf{\textbf{false}} & \text{if } bv = a \\ \textsf{\textbf{true}} & \text{otherwise} \end{cases}$$

2.2.14 The tostr Operator.

The **tostr** unary operator converts a primitive value to a string. The translator guarantees that bv is never an address.

$$\textsf{\textbf{tostr}}\ bv = \begin{cases} n.\mathit{toString} & \text{if } bv = n \\ \texttt{"true"} & \text{if } bv = \textsf{\textbf{true}} \\ \texttt{"false"} & \text{if } bv = \textsf{\textbf{false}} \\ \mathit{str} & \text{if } bv = \mathit{str} \\ \texttt{"null"} & \text{if } bv = \textsf{\textbf{null}} \\ \texttt{"undefined"} & \text{if } bv = \textsf{\textbf{undef}} \end{cases}$$

2.2.15 The tonum Operator.

The **tonum** unary operator converts a primitive value to a number. The translator guarantees that bv is never an address.

$$\textsf{\textbf{tonum}}\ bv = \begin{cases} n & \text{if } bv = n \\ 1 & \text{if } bv = \textsf{\textbf{true}} \\ 0 & \text{if } bv = \textsf{\textbf{false}} \\ \mathit{str}.\mathit{toDouble} & \text{if } bv = \mathit{str},\ \mathit{str} \text{ represents a valid number} \\ \texttt{NaN} & \text{if } bv = \mathit{str},\ \mathit{str} \text{ doesn't represent a valid number} \\ 0 & \text{if } bv = \textsf{\textbf{null}} \\ \texttt{NaN} & \text{if } bv = \textsf{\textbf{undef}} \end{cases}$$

2.3 Concrete State Transition Rules

Note: The ECMA standard states that implementations evaluating **for** $x\,e\,s$ where $[\![e]\!] \in \{\textsf{\textbf{null}}, \textsf{\textbf{undef}}\}$ should skip the loop entirely, but that they may choose instead to throw a `"TypeError"` exception. We conform to the standard, skipping the loop. One might question why the **finK** continuation is used when entering a **finally** statement with a normal value, since it's just thrown away without being used; this will turn out to become important in the abstract semantics.

Table 1: The concrete semantics transition function $\mathcal{F}$. Each rule is describes a transition relation from one concrete state $\langle t, \rho, \sigma, \kappa \rangle$ to the next concrete state $\langle t_{new}, \rho_{new}, \sigma_{new}, \kappa_{new} \rangle$. We use :: to indicate concatenation of sequences and _ to indicate "don't care".

#	t	Premises	t_{new}	ρ_{new}	σ_{new}	κ_{new}
1	$\textsf{decl}\ \overrightarrow{x_i = e_i}\ \textsf{in}\ s$	$\overrightarrow{bv_i} = \llbracket \vec{e_i} \rrbracket, (\sigma', \vec{a_i}) = \texttt{alloc}(\sigma, \overrightarrow{bv_i}),\ \rho' = \rho[\overrightarrow{x_i \mapsto a_i}]$	s	ρ'	σ'	κ
2	$s :: \vec{s_i}$		s	ρ	σ	$\textsf{seqK}\ \vec{s_i}\ \kappa$
3	$\textsf{if}\ e\ s_1\ s_2$	$\textsf{true} = \llbracket e \rrbracket$	s_1	ρ	σ	κ
4	$\textsf{if}\ e\ s_1\ s_2$	$\textsf{false} = \llbracket e \rrbracket$	s_2	ρ	σ	κ
5	$x := e$	$bv = \llbracket e \rrbracket,\ \rho(x) = a$	bv	ρ	$\sigma[a \mapsto bv]$	κ
6	$\textsf{while}\ e\ s$	$\textsf{true} = \llbracket e \rrbracket$	s	ρ	σ	$\textsf{whileK}\ e\ s\ \kappa$
7	$\textsf{while}\ e\ s$	$\textsf{false} = \llbracket e \rrbracket$	$\textsf{undef}$	ρ	σ	κ
8	$x := \textsf{newfun}\ m\ n$	$a = \rho(x),\ (\sigma', a') = \texttt{allocFun}((\rho, m), n, \sigma)$	a'	ρ	$\sigma'[a \mapsto a']$	κ
9	$x := \textsf{new}\ e_1(e_2)$	$(\sigma', a) = \texttt{allocObj}(\llbracket e_1 \rrbracket, \sigma)$, $\sigma'' = \texttt{setConstr}(\sigma'[\rho(x) \mapsto a], \llbracket e_2 \rrbracket)$, $\varsigma = \texttt{applyClo}(\llbracket e_1 \rrbracket, a, \llbracket e_2 \rrbracket, x, \rho, \sigma'', \kappa)$	$\pi_t(\varsigma)$	$\pi_\rho(\varsigma)$	$\pi_\sigma(\varsigma)$	$\pi_\kappa(\varsigma)$
10	$x := \textsf{toobj}\ e$	$(v, \sigma') = \texttt{toObj}(\llbracket e \rrbracket, x, \rho, \sigma)$	v	ρ	σ'	κ
11	$e_1.e_2 := e_3$	$(v, \sigma') = \texttt{updateObj}(\llbracket e_1 \rrbracket, \llbracket e_2 \rrbracket, \llbracket e_3 \rrbracket, \sigma)$	v	ρ	σ'	κ
12	$x := \textsf{del}\ e_1.e_2$	$(v, \sigma') = \texttt{delete}(\llbracket e_1 \rrbracket, \llbracket e_2 \rrbracket, x, \rho, \sigma)$	v	ρ	σ'	κ
13	$\textsf{try-catch-fin}\ s_1\ x\ s_2\ s_3$		s_1	ρ	σ	$\textsf{tryK}\ x\ s_2\ s_3\ \kappa$
14	$\textsf{throw}\ e$	$bv = \llbracket e \rrbracket$	$\textsf{exc}\ bv$	ρ	σ	κ
15	$\textsf{jump}\ \ell\ e$	$bv = \llbracket e \rrbracket$	$\textsf{jmp}\ \ell\ bv$	ρ	σ	κ
16	$\ell\ s$		s	ρ	σ	$\textsf{lblK}\ \ell\ \kappa$
17	$x := e_1(e_2, e_3)$	$\varsigma = \texttt{applyClo}(\llbracket e_1 \rrbracket, \llbracket e_2 \rrbracket, \llbracket e_3 \rrbracket, x, \rho, \sigma, \kappa)$	$\pi_t(\varsigma)$	$\pi_\rho(\varsigma)$	$\pi_\sigma(\varsigma)$	$\pi_\kappa(\varsigma)$
18	$\textsf{for}\ x\ e\ s$	$str :: \overrightarrow{str_i} = \texttt{objKeys}(\llbracket e \rrbracket, \sigma),\ a = \rho(x)$	s	ρ	$\sigma[a \mapsto str]$	$\textsf{forK}\ \overrightarrow{str_i}\ x\ s\ \kappa$
19	$\textsf{for}\ x\ e\ s$	$\texttt{objKeys}(\llbracket e \rrbracket, \sigma) = \emptyset$	$\textsf{undef}$	ρ	σ	κ
20	bv	$\kappa = \textsf{seqK}\ s :: \vec{s_i}\ \kappa_c$	s	ρ	σ	$\textsf{seqK}\ \vec{s_i}\ \kappa_c$
21	bv	$\kappa = \textsf{seqK}\ \emptyset\ \kappa_c$	bv	ρ	σ	κ_c
22	bv	$\kappa = \textsf{whileK}\ e\ s\ \kappa_c,\ \textsf{true} = \llbracket e \rrbracket$	s	ρ	σ	κ
23	bv	$\kappa = \textsf{whileK}\ e\ s\ \kappa_c,\ \textsf{false} = \llbracket e \rrbracket$	$\textsf{undef}$	ρ	σ	κ_c
24	bv	$\kappa = \textsf{forK}\ str :: \overrightarrow{str_i}\ x\ s\ \kappa_c,\ a = \rho(x)$	s	ρ	$\sigma[a \mapsto str]$	$\textsf{forK}\ \overrightarrow{str_i}\ x\ s\ \kappa_c$
25	bv	$\kappa = \textsf{forK}\ \emptyset\ x\ s\ \kappa_c$	$\textsf{undef}$	ρ	σ	κ_c
26	bv	$\kappa = \textsf{retK}\ x\ \rho_c\ \kappa_c\ ctxt,\ a = \rho_c(x), (ctxt = \textsf{call} \vee bv = a')$	bv	ρ_c	$\sigma[a \mapsto bv]$	κ_c
27	bv	$\kappa = \textsf{retK}\ x\ \rho_c\ \kappa_c\ \textsf{ctor},\ bv \neq a$	$\llbracket x \rrbracket$	ρ_c	σ	κ_c
28	ev	$\kappa = \textsf{retK}\ x\ \rho_c\ \kappa_c\ \_$	ev	ρ_c	σ	κ_c
29	bv	$\kappa = \textsf{tryK}\ x\ s_c\ s_f\ \kappa_c$	s_f	ρ	σ	$\textsf{finK}\ \textsf{undef}\ \kappa_c$
30	jv	$\kappa = \textsf{tryK}\ x\ s_c\ s_f\ \kappa_c$	s_f	ρ	σ	$\textsf{finK}\ jv\ \kappa_c$
31	$\textsf{exc}\ bv$	$\kappa = \textsf{tryK}\ x\ s_c\ s_f\ \kappa_c,\ a = \rho(x)$	s_c	ρ	$\sigma[a \mapsto bv]$	$\textsf{catchK}\ s_f\ \kappa_c$
32	bv	$\kappa = \textsf{catchK}\ s_f\ \kappa_c$	s_f	ρ	σ	κ_c
33	v	$\kappa = \textsf{catchK}\ s_f\ \kappa_c,\ v \in EValue \cup JValue$	s_f	ρ	σ	$\textsf{finK}\ v\ \kappa_c$
34	bv	$\kappa = \textsf{finK}\ v\ \kappa_c,\ v' = v \in EValue \cup JValue\ ?\ v : bv$	v'	ρ	σ	κ_c
35	v	$\kappa = \textsf{lblK}\ \ell\ \kappa_c,\ (v = \textsf{jmp}\ \ell\ bv \vee v = bv)$	bv	ρ	σ	κ_c
36	$\textsf{jmp}\ \ell\ bv$	$\kappa = \textsf{lblK}\ \ell'\ \kappa_c,\ \ell \neq \ell'$	$\textsf{jmp}\ \ell\ bv$	ρ	σ	κ_c
37	v	$\kappa \notin \texttt{specialK}(v),\ v \in EValue \cup JValue$	v	ρ	σ	$\texttt{nextK}(\kappa)$

2.4 Concrete Helper Functions

We define the concrete helper functions used by the previous sections. The functions are listed in alphabetical order. We use the notation $\mathcal{O}(\cdot)$ to indicate an "Option" type: essentially a set guaranteed to contain zero or one values.

2.4.1 `alloc`

`alloc` takes a store and a list of values and returns a new store and a list of addresses such that those addresses are fresh and the new store maps the new addresses to the given values.

$$\texttt{alloc} \in \mathit{Store} \times \overrightarrow{\mathit{BValue}_i} \rightarrow \mathit{Store} \times \overrightarrow{\mathit{Address}_i}$$
$$\begin{aligned}
&\texttt{alloc}(\sigma, \vec{v_i}) = (\sigma', \vec{a_i}) \quad \text{where} \\
&\qquad \vec{a_i} \cap \texttt{dom}(\sigma) = \emptyset \\
&\qquad \sigma' = \sigma[\overrightarrow{a_i \mapsto v_i}]
\end{aligned}$$

2.4.2 `allocFun`

`allocFun` allocates a function object into the store, setting its properties appropriately. `Function_prototype_Addr` is defined in `builtin.pdf`.

$$\texttt{allocFun} \in \mathit{Closure} \times \mathit{Num} \times \mathit{Store} \rightarrow \mathit{Store} \times \mathit{Address}$$
$$\begin{aligned}
&\texttt{allocFun}(\mathit{clo}, n, \sigma) = (\sigma', a') \quad \text{where} \\
&\qquad a' \notin \texttt{dom}(\sigma) \\
&\qquad \mathit{internal} = [\texttt{"proto"} \mapsto \texttt{Function_prototype_Addr}, \texttt{"class"} \mapsto \textbf{function}, \texttt{"code"} \mapsto \mathit{clo}] \\
&\qquad \mathit{external} = [\texttt{"length"} \mapsto n] \\
&\qquad \sigma' = \sigma[a' \mapsto (\mathit{external}, \mathit{internal})]
\end{aligned}$$

2.4.3 `allocObj`

`allocObj` allocates objects into the store; the object's class is based on the constructor function object's address, which is the first parameter. It makes use of the helper function `classFromAddress` and the value `Object_prototype_Addr`, both defined in `builtin.pdf`.

$$\texttt{allocObj} \in \mathit{Address} \times \mathit{Store} \rightarrow \mathit{Store} \times \mathit{Address}$$
$$\begin{aligned}
&\texttt{allocObj}(a, \sigma) = (\sigma', a') \quad \text{where} \\
&\qquad a' \notin \texttt{dom}(\sigma) \\
&\qquad c = \texttt{classFromAddress}(a) \\
&\qquad a'' = \begin{cases} \pi_1(\sigma(a))(\texttt{"prototype"}) & \text{if } \pi_1(\sigma(a))(\texttt{"prototype"}) \in \mathit{Address} \\ \texttt{Object_prototype_Addr} & \text{otherwise} \end{cases} \\
&\qquad \mathit{internal} = [\texttt{"proto"} \mapsto a'', \texttt{"class"} \mapsto c] \\
&\qquad \sigma' = \sigma[a' \mapsto (\emptyset, \mathit{internal})]
\end{aligned}$$

2.4.4 `applyClo`

`applyClo` is used to make method calls. It must check that the function expression evaluates to an object whose class is `callable`. The predicate `callable` is given in `builtin.pdf`, `callable` maps each class that has an internal `"code"` field (like the **function** class) to **true**, and the remaining classes to **false**. The translator guarantees that the two arguments evaluate to addresses where the first is the `self` pointer and the second is a pointer to the `args` array containing the call's arguments. If this check passes then `applyClo` returns a new state containing the callee's body, the new environment mapping the parameters to their addresses, and the new store mapping those addresses to the corresponding arguments' values. If this check fails then `applyClo` returns a new state containing an exception. For convenience we use *body* to refer to either a statement $s \in \mathit{Stmt}$ or a declaration $d \in \mathit{Decl}$, since the callee's body can be either one and we do the same thing in either case.

$\texttt{applyClo} \in BValue \times BValue \times BValue \times Variable \times Env \times Store \times Kont \to State$

$\texttt{applyClo}(bv_1, bv_2, bv_3, x, \rho, \sigma, \kappa) = \varsigma \quad$ where:

$$\varsigma = \begin{cases} (body,\ \rho'_c,\ \sigma', \kappa') & \text{if } bv_1 = a_1,\ \texttt{callable}(\texttt{getClass}(\sigma(a_1))),\ (\rho_c, (\texttt{self}, \texttt{args}) \Rightarrow body) = \pi_2(\sigma(a_1))(\texttt{"code"}) \\ (\textsf{exc}\ \texttt{"TypeError"}, \rho, \sigma, \kappa) & \text{otherwise.} \end{cases}$$

where

$$(\sigma',\ \{a'_2, a'_3\}) = \texttt{alloc}(\sigma, \{bv_2, bv_3\})$$

$$\rho'_c = \rho_c[\texttt{self} \mapsto a'_2,\ \texttt{args} \mapsto a'_3]$$

$$ctxt = \begin{cases} \textsf{ctor} & \text{if } a_3 = bv_3,\ \texttt{"constructor"} \in \texttt{dom}(\pi_2(\sigma(a_3))). \\ \textsf{call} & \text{otherwise.} \end{cases}$$

$$\kappa' = \textsf{retK}\ x\ \rho\ \kappa\ ctxt$$

2.4.5 `charAt`

`charAt` takes a primitive string and an index and returns the character at that index as a string. The behavior is undefined if the index is larger than $\texttt{len}(str)$.

$\texttt{charAt} \in String \times \mathbb{N} \to String$

2.4.6 `delete`

`delete` removes a property from an object as long as that property does not have the **nodelete** attribute (we implicitly ignore such properties when computing the object's external map's domain with `dom`). If the deletion was successful `delete` assigns **true** to the given variable, otherwise it assigns **false** (unless the attempted deletion raises an exception, in which case the variable remains unchanged).

$\texttt{delete} \in BValue \times BValue \times Variable \times Env \times Store \to Value \times Store$

$\texttt{delete}(bv_1, bv_2, x, \rho, \sigma) =$

$$\begin{cases} (\textsf{exc}\ \texttt{"TypeError"}, \sigma) & \text{if } bv_1 \in \{\textsf{null}, \textsf{undef}\} \\ (\textsf{undef}, \sigma') & \text{if } a = bv_1,\ str = bv_2,\ str \in \texttt{dom}(\sigma(a)) \\ (\textsf{undef}, \sigma'') & \text{otherwise} \end{cases}$$

where

$$\sigma' = \sigma[a \mapsto \sigma(a) - str,\ \rho(x) \mapsto \textsf{true}]$$

$$\sigma'' = \sigma[\rho(x) \mapsto \textsf{false}]$$

2.4.7 `getProto`

`getProto` retrieves the `"proto"` property from an object's internal map.

$\texttt{getProto} \in Object \to Address$

$\texttt{getProto}(o) = \pi_2(o)(\texttt{"proto"})$

2.4.8 `getClass`

`getClass` retrieves the `"class"` property from an object's internal map.

$\texttt{getClass} \in Object \to Class$

$\texttt{getClass}(o) = \pi_2(o)(\texttt{"class"})$

2.4.9 `initState`

`initState` takes a program and returns the initial concrete state, containing the initial environment and store.

$\texttt{initState} \in \mathit{Decl} \rightarrow \mathit{State}$

$\texttt{initState}(d) = (d,\ \rho,\ \sigma,\ \textsf{\textbf{haltK}})$

where ρ and σ are created as described in `builtin.pdf`.

2.4.10 `len`

`len` gives the length of a string primitive.

$\texttt{len} \in \mathit{String} \rightarrow \mathbb{N}$

2.4.11 `lookup`

`lookup` looks up the value of a property in an object, going up the prototype chain if necessary.

$\texttt{lookup} \in \mathit{Address} \times \mathit{String} \times \mathit{Store} \rightarrow \mathit{BValue}$

$\texttt{lookup}(a, \mathit{str}, \sigma) =$

$$\begin{cases} o(\mathit{str}) & \text{if } \mathit{str} \in \texttt{dom}(o) \\ \texttt{lookup}(a', \mathit{str}, \sigma) & \text{if } \mathit{str} \notin \texttt{dom}(o),\ a' = \texttt{getProto}(o) \\ \textsf{\textbf{undef}} & \text{otherwise} \end{cases}$$

where $o = \sigma(a)$

2.4.12 `nextK`

`nextK` takes continuations containing other continuations (i.e., any continuation other than **haltK**) and returns the enclosed continuation. It is, in essence, "popping" the continuation stack.

$\texttt{nextK} \in \mathit{Kont} \setminus \{\textsf{\textbf{haltK}}\} \rightarrow \mathit{Kont}$

$\texttt{nextK}(\kappa) = \kappa_c \quad \text{where}$

$\kappa = \langle \mathit{name} \rangle \ldots \kappa_c$

2.4.13 `objKeys`

`objKeys` returns the set of enumerable properties in an object as a sequence in arbitrary order (enumerable properties are all properties not listed as having the **noenum** attribute; we implicitly omit such properties when computing the object's external map's domain with `dom`). If given a non-object, or if the given object doesn't have any enumerable properties, then `objKeys` returns an empty sequence.

$\texttt{objKeys} \in \mathit{BValue} \times \mathit{Store} \rightarrow \overrightarrow{\mathit{String}_i}$

$\texttt{objKeys}(bv, \sigma) =$

$$\begin{cases} \emptyset & \text{if } bv \notin \mathit{Address} \vee (bv = a,\ \texttt{dom}(\sigma(a)) = \emptyset) \\ \texttt{dom}(\sigma(a)) & \text{if } bv = a,\ \texttt{dom}(\sigma(a)) \neq \emptyset \end{cases}$$

2.4.14 `setConstr`

`setConstr` sets the internal `"constructor"` property of the object referenced by the second parameter to **true**. This enables the callee function in the **new** $e_1(e_2)$ rule to know that it was called as a JavaScript constructor.

$\texttt{setConstr} \in \mathit{Store} \times \mathit{Address} \rightarrow \mathit{Store}$

$\texttt{setConstr}(\sigma, a) = \sigma' \quad \text{where}$

$$\begin{aligned}
&\mathit{external} = \pi_1(\sigma(a)) \\
&\mathit{internal} = \pi_2(\sigma(a)) \\
&\sigma' = \sigma[a \mapsto (\mathit{external}, \mathit{internal}[\texttt{"constructor"} \mapsto \textbf{true}])]
\end{aligned}$$

2.4.15 `specialK`

`specialK` is used by the default exceptional/jump value propagation rule that pops the continuation stack until it finds a relevant continuation. Given a value, `specialK` returns the set of continuations that should not be popped when propagating that type of value because there are other rules that handle them.

$\texttt{specialK} \in \mathit{Value} \rightarrow \mathcal{P}(\mathit{Kont})$

$\texttt{specialK}(v) =$

$$\begin{cases} \mathit{Kont} & \text{if } v \in \mathit{BValue} \\ \{\textbf{haltK}, \textbf{tryK}, \textbf{catchK}, \textbf{retK}\} & \text{if } v \in \mathit{EValue} \\ \{\textbf{haltK}, \textbf{tryK}, \textbf{catchK}, \textbf{lblK}\} & \text{if } v \in \mathit{JValue} \end{cases}$$

2.4.16 `toObj`

`toObj` attempts to convert a value into an object, allocating a new object if necessary. The addresses `Number_Addr`, `Boolean_Addr`, and `String_Addr` are placed in store by `initState` and are globally available.

$\texttt{toObj} \in \mathit{BValue} \times \mathit{Variable} \times \mathit{Env} \times \mathit{Store} \rightarrow \mathit{Value} \times \mathit{Store}$

$\texttt{toObj}(bv, x, \rho, \sigma) =$

$$\begin{cases} (\textbf{exc}\ \texttt{"TypeError"}, \sigma) & \text{if } bv \in \{\textbf{null}, \textbf{undef}\} \\ (bv, \sigma[\rho(x) \mapsto bv]) & \text{if } bv \in \mathit{Address} \\ (a', \sigma''[\rho(x) \mapsto a']) & \text{otherwise} \end{cases}$$

where

$$a = \begin{cases} \texttt{Number\_Addr} & \text{if } bv \in \mathit{Num} \\ \texttt{Boolean\_Addr} & \text{if } bv \in \mathit{Bool} \\ \texttt{String\_Addr} & \text{if } bv \in \mathit{String} \end{cases}$$

$$\begin{aligned}
&(a', \sigma') = \texttt{allocObj}(a, \sigma) \\
&\mathit{external} = \pi_1(\sigma'(a')) \\
&\mathit{internal} = \pi_2(\sigma'(a'))
\end{aligned}$$

$$\sigma'' = \begin{cases} \sigma'[a' \mapsto (\mathit{external}[\texttt{"}i\texttt{"} \mapsto \texttt{charAt}(str, i) \mid 0 \leq i < \texttt{len}(str)][\texttt{"length"} \mapsto \texttt{len}(str)] & \text{if } bv = str \\ \sigma'[a' \mapsto (\mathit{external}, \mathit{internal}[\texttt{"value"} \mapsto bv])] & \text{otherwise} \end{cases}$$

2.4.17 `updateObj`

`updateObj` attempts to update the value of a property in an object: the first argument is the object, the second is the property to update, the third is the new value. If the given object is actually **null** or **undef** then `updateObj` raises an exception. If the property being updated has the attribute **noupdate** then `updateObj` silently fails (i.e., returns bv_3 but doesn't actually update the object). The translator guarantees that the first argument is either the address of an object or **null** or **undef**, and that the second argument is a string. Array objects also have special rules involving property updates that we enforce here.

$\texttt{updateObj} \in \mathit{BValue} \times \mathit{BValue} \times \mathit{BValue} \times \mathit{Store} \rightarrow \mathit{Value} \times \mathit{Store}$

$\texttt{updateObj}(bv_1, bv_2, bv_3, \sigma) =$

$$\begin{cases} (\textsf{exc}\ \texttt{"TypeError"}, \sigma) & \text{if } bv_1 \in \{\textsf{null}, \textsf{undef}\} \\ (\textsf{exc}\ \texttt{"RangeError"}, \sigma) & \text{if } a = bv_1,\ str = bv_2,\ \texttt{getClass}(\sigma(a)) = \textsf{array},\ str = \texttt{"length"},\ \neg u32?(bv_3) \\ (bv_3, \sigma_1) & \text{if } a = bv_1,\ str = bv_2,\ \texttt{getClass}(\sigma(a)) = \textsf{array},\ str = \texttt{"length"},\ u32?(bv_3) \\ (bv_3, \sigma_2) & \text{if } a = bv_1,\ str = bv_2,\ \texttt{getClass}(\sigma(a)) = \textsf{array},\ str \neq \texttt{"length"},\ u32?(\textsf{tonum}\ str) \\ (bv_3, \sigma_3) & \text{if } a = bv_1,\ str = bv_2,\ \texttt{getClass}(\sigma(a)) \neq \textsf{array} \vee (str \neq \texttt{"length"},\ \neg u32?(\textsf{tonum}\ str)) \end{cases}$$

where

$u32?(bv) = bv$ is an unsigned 32-bit integer

$o = \sigma(a)[str \mapsto bv_3] - \{\ str \ \mid\ str \in \texttt{dom}(\sigma(a)),\ u32?(\textsf{tonum}\ str),\ bv_3 \leq \textsf{tonum}\ str\ \}$

$$\sigma_1 = \begin{cases} \sigma_3 & \text{if } \sigma(a)(\texttt{"length"}) \leq bv_3 \\ \sigma[a \mapsto o] & \text{otherwise} \end{cases}$$

$$\sigma_2 = \begin{cases} \sigma_3 & \text{if } \sigma(a)(\texttt{"length"}) > bv_3 \vee bv_3 = 2^{31} - 1 \\ \sigma_3[a \mapsto \sigma_3(a)[\texttt{"length"} \mapsto bv_3 + 1]] & \text{otherwise} \end{cases}$$

$\sigma_3 = \sigma[a \mapsto \sigma(a)[str \mapsto bv_3]]$

3 Abstract Semantics

In this section we describe the following: (1) the abstract semantic domains; (2) the abstraction and concretization functions relating the abstract and concrete semantic domains; (3) the abstract evaluation of expressions; (4) the abstract state transition rules; and (5) the abstract helper functions used by the above descriptions.

Note: The abstract semantics is specified here without any form of control-flow sensitivity—a straight implementation of this semantics would yield an intractable path-sensitive analysis. We apply the widening operator method from Hardekopf et al, "Widening for Control-Flow" on top of the semantics given here to create an analysis with tunable control-flow sensitivity. That method gives a modular, tunable way to specify when states should be merged. In order to maximize precision while remaining tractable, we only potentially merge states at certain points in the abstract execution where multiple control-flow paths join together (e.g., after both branches of an **if** statement). We specify the exact set of potential points by having the translator add special **merge** statements to the desugared program; these statements are noops in terms of the program, but signal the abstract interpreter to apply the widening operator. Since the statements are noops and only relevant to the widening operator, we ignore them in this specification.

3.1 Abstract Semantic Domains

$$\hat{n} \in Num^\sharp \qquad \widehat{str} \in String^\sharp \qquad \hat{a} \in Address^\sharp \qquad \hat{\odot} \in UnaryOp^\sharp \qquad \hat{\oplus} \in BinaryOp^\sharp$$

$$\begin{aligned}
\hat{\varsigma} \in State^\sharp &= Term^\sharp \times Env^\sharp \times Store^\sharp \times Kont^\sharp \\
\hat{t} \in Term^\sharp &= Decl + Stmt + Value^\sharp \\
\hat{\rho} \in Env^\sharp &= Variable \to \mathcal{P}(Address^\sharp) \\
\hat{\sigma} \in Store^\sharp &= Address^\sharp \to (BValue^\sharp + Object^\sharp + \mathcal{P}(Kont^\sharp)) \\
\widehat{bv} \in BValue^\sharp &= Num^\sharp \times \mathcal{P}(Bool) \times String^\sharp \times \mathcal{P}(Address^\sharp) \times \mathcal{P}(\{\textbf{null}\}) \times \mathcal{P}(\{\textbf{undef}\}) \\
\hat{o} \in Object^\sharp &= (String^\sharp \to BValue^\sharp) \times (String \to (BValue^\sharp + Class + \mathcal{P}(Closure^\sharp))) \times \mathcal{P}(String) \\
\widehat{clo} \in Closure^\sharp &= Env^\sharp \times Method \\
\widehat{ev} \in EValue^\sharp &= \textbf{exc}\ BValue^\sharp \\
\widehat{jv} \in JValue^\sharp &= \textbf{jmp}\ Label \times BValue^\sharp \\
\hat{v} \in Value^\sharp &= BValue^\sharp + EValue^\sharp + JValue^\sharp \\
\hat{\kappa} \in Kont^\sharp &= \widehat{\textbf{haltK}} + \\
&\quad \widehat{\textbf{seqK}}\ \overrightarrow{Stmt}_i\ Kont^\sharp + \\
&\quad \widehat{\textbf{whileK}}\ Exp\ Stmt\ Kont^\sharp + \\
&\quad \widehat{\textbf{forK}}\ \overrightarrow{String^\sharp}_i\ Variable\ Stmt\ Kont^\sharp + \\
&\quad \widehat{\textbf{retK}}\ Variable\ Env^\sharp\ Kont^\sharp\ \{\textbf{ctor}, \textbf{call}\} + \\
&\quad \widehat{\textbf{tryK}}\ Variable\ Stmt\ Stmt\ Kont^\sharp + \\
&\quad \widehat{\textbf{catchK}}\ Stmt\ Kont^\sharp + \\
&\quad \widehat{\textbf{finK}}\ \mathcal{P}(Value^\sharp)\ Kont^\sharp + \\
&\quad \widehat{\textbf{lblK}}\ Label\ Kont^\sharp \\
&\quad \widehat{\textbf{addrK}}\ Address^\sharp
\end{aligned}$$

Environments map to sets of addresses because under some forms of control-flow sensitivity, two states with different environments can be joined; thus we need a lattice of environments. The store's co-domain now includes sets of continuations because the $\widehat{\textbf{retK}}$ continuations are store-allocated to allow for finite abstraction of recursive functions (see Van Horn and Might's "Abstracting Abstract Machines"). The $\widehat{\textbf{addrK}}$ continuation holds the address of the $\widehat{\textbf{retK}}$ continuations. Base values are a tuple instead of a sum: because JavaScript is dynamically typed, we cannot restrict a variable's value to only one type, and because we are over-approximating the computation, an abstract value can have multiple types at the same time. An object now includes a set of definitely-present properties; these properties and the domain of the internal map are *String* instead of $String^\sharp$ because they are exactly known. As before, we abuse notation by treating an abstract object $\hat{o}$ as a map to implicitly mean its programmer-visible map.

Note: For the abstract store, the abstract allocation functions $\texttt{alloc}^\sharp$ and $\texttt{allocObj}^\sharp$ guarantee that the set of addresses that can map to $BValue^\sharp$, the sets of addresses that can map to each class of $Object^\sharp$, and the set of addresses that can map to $\mathcal{P}(Kont^\sharp)$ are all disjoint.

3.1.1 Property Attributes

We use the same method to handle property attributes as for the concrete semantics. This is sound as long as we detect any attempt to use the API to modify property attributes.

3.1.2 The $Num^\sharp$, $String^\sharp$, and $Address^\sharp$ Domains

We deliberately leave these domains unspecified in order to allow for many possible abstractions, with the caveat that the address domain must be finite. The default domains are implemented in the abstract interpreter; see the code for details. The interpreter is implemented in such a way that replacing a default domain with a different one involves replacing a single class, without needing to change the rest of the code.

3.1.3 The $Object^\sharp$ Domain

Interpreting the external map of an abstract object is a bit tricky. In the external map, a property $\widehat{str} \mapsto \widehat{bv}$ is interpreted to mean $\exists str \in \gamma_{str}(\widehat{str})\,.\, str \mapsto \widehat{bv}$. If the property name is inexact (i.e., $|\gamma_{str}(\widehat{str})| > 1$) then for any particular exact property name $str \in \gamma_{str}(\widehat{str})$ we do not know whether it is present or not. If the property name is exact but the property is not definitely present (i.e., not a member of the definitely-present set) then we still do not know if it is present or not. Only if the property name is exact and the property is definitely present do we know that it is really there. For these reasons, abstract property update and lookup are complicated; we formally describe them in Section 3.4, but we give an informal explanation here to aid understanding.

Update a property. We keep the update and join operations simple and cheap by trading-off a more complex lookup. There are two cases to consider:

- **Strong update of $\widehat{str}$ to $\widehat{bv}$.** We are strongly updating a property, which implies that $\widehat{str}$ is exact. We update the external map so that $\widehat{str} \mapsto \widehat{bv}$, and we place $\widehat{str}$ in the set of definitely-present properties.
- **Weak update of $\widehat{str}$ to bv.** If $\widehat{str}$ is not already present, we update the external map so that $\widehat{str} \mapsto \widehat{bv}$. If $\widehat{str}$ is present with value $\widehat{bv}'$, we update the external map so that $\widehat{str} \mapsto \widehat{bv} \sqcup \widehat{bv}'$.

Lookup a property. Because of the simple update procedure, lookup of a property $\widehat{str}$ requires some extra effort. There are three cases to consider:

- **$\widehat{str}$ is definitely present.** This implies that $\widehat{str}$ is exact. Let $\overline{\widehat{str}}$ be the set of all properties present in the object that over-approximate $\widehat{str}$ (including $\widehat{str}$ itself). We join all of those properties' values together and return the result. We must join the values because updates to less-precise properties may have actually been updates to $\widehat{str}$.
- **$\widehat{str}$ is definitely not present.** We recursively look up the object's prototype chain as described in Section 3.4.
- **$\widehat{str}$ is possibly present.** Let $\overline{\widehat{str}}$ be the set of all properties present in the object that are comparable to $\widehat{str}$ (including $\widehat{str}$ itself). We join all of those properties' values together; call the result $\widehat{bv}$. We then join $\widehat{bv}$ with the result of recursively looking up the prototype chain. We need all comparable properties' values because updates to less-precise properties may have actually been updates to $\widehat{str}$, and updates to more-precise properties should be included in the less-precise $\widehat{str}$'s value.

3.2 Abstract Expression Evaluation

Note: In Sections 3.2, 3.3, and 3.4 we use the notation $[\![\cdot]\!]$ to mean $\eta^\sharp(\cdot, \hat{\rho}, \hat{\sigma})$ when $\hat{\rho}$ and $\hat{\sigma}$ are obvious from the context. We often implicitly lift non-sets to singleton sets when required by a function's signature, in order to make the notation less cluttered. We define an evaluator for pure expressions $e \in \mathit{Exp}$:

$$\eta^\sharp : \mathit{Exp} \times \mathit{Env}^\sharp \times \mathit{Store}^\sharp \to \mathit{BValue}^\sharp$$

$$\eta^\sharp(e, \hat{\rho}, \hat{\sigma}) =$$

$$\begin{cases} \mathtt{inject}^\sharp(e) & \text{if } e \in \mathit{Bool} \cup \{\textsf{\textbf{null}}, \textsf{\textbf{undef}}\} \\ \mathtt{inject}^\sharp(\alpha_n(e)) & \text{if } e \in \mathit{Num} \\ \mathtt{inject}^\sharp(\alpha_{str}(e)) & \text{if } e \in \mathit{String} \\ \bigsqcup\limits_{\hat{a} \in \hat{\rho}(e)} \hat{\sigma}(\hat{a}) & \text{if } e \in \mathit{Variable} \\ [\![e_1]\!] \mathbin{\hat{\oplus}} [\![e_2]\!] & \text{if } e = e_1 \oplus e_2 \\ \hat{\odot}[\![e']\!] & \text{if } e = \odot e' \end{cases}$$

3.2.1 Numeric Binary Operators.

For binary operators $\hat{\oplus} \in \{+, -, \times, \div, \%, \ll, \gg, \ggg, <, \leq, \&, |, \veebar\}$:

$$\widehat{bv}_1 \mathbin{\hat{\oplus}} \widehat{bv}_2 = \mathtt{inject}^\sharp(\pi_{\hat{n}}(\widehat{bv}_1) \mathbin{\hat{\oplus}} \pi_{\hat{n}}(\widehat{bv}_2))$$

where the operators in $\hat{\oplus}$ are defined on and return $\mathit{Num}^\sharp$ and are specific to a given numeric abstract domain.

3.2.2 String Binary Operators.

For binary operators $\hat{\oplus} \in \{+\!\!+, \prec, \preceq\}$:

$$\widehat{bv}_1 \mathbin{\hat{\oplus}} \widehat{bv}_2 = \mathtt{inject}^\sharp(\pi_{\widehat{str}}(\widehat{bv}_1) \mathbin{\hat{\oplus}} \pi_{\widehat{str}}(\widehat{bv}_2))$$

where the operators in $\hat{\oplus}$ are defined on and return $\mathit{String}^\sharp$ and are specific to a given string abstract domain.

3.2.3 Boolean Binary Operators

For binary operators $\{\textsf{\textbf{and}}, \textsf{\textbf{or}}\}$:

$$\widehat{bv}_1 \; \textsf{\textbf{and}} \; \widehat{bv}_2 = \begin{cases} \emptyset & \text{if } \pi_{\hat{b}}(\widehat{bv}_1) = \emptyset \vee \pi_{\hat{b}}(\widehat{bv}_2) = \emptyset \\ \textsf{\textbf{true}} & \text{if } \pi_{\hat{b}}(\widehat{bv}_1) = \{\textsf{\textbf{true}}\},\ \pi_{\hat{b}}(\widehat{bv}_2) = \{\textsf{\textbf{true}}\} \\ \textsf{\textbf{false}} & \text{if } \pi_{\hat{b}}(\widehat{bv}_1) = \{\textsf{\textbf{false}}\} \vee \pi_{\hat{b}}(\widehat{bv}_2) = \{\textsf{\textbf{false}}\} \\ \{\textsf{\textbf{true}}, \textsf{\textbf{false}}\} & \text{otherwise} \end{cases}$$

$$\widehat{bv}_1 \; \textsf{\textbf{or}} \; \widehat{bv}_2 = \begin{cases} \emptyset & \text{if } \pi_{\hat{b}}(\widehat{bv}_1) = \emptyset \vee \pi_{\hat{b}}(\widehat{bv}_2) = \emptyset \\ \textsf{\textbf{true}} & \text{if } \pi_{\hat{b}}(\widehat{bv}_1) = \{\textsf{\textbf{true}}\} \vee \pi_{\hat{b}}(\widehat{bv}_2) = \{\textsf{\textbf{true}}\} \\ \textsf{\textbf{false}} & \text{if } \pi_{\hat{b}}(\widehat{bv}_1) = \{\textsf{\textbf{false}}\},\ \pi_{\hat{b}}(\widehat{bv}_2) = \{\textsf{\textbf{false}}\} \\ \{\textsf{\textbf{true}}, \textsf{\textbf{false}}\} & \text{otherwise} \end{cases}$$

3.2.4 Strict Equality Operator.

For the binary operator $\equiv$:

$$\widehat{bv}_1 \equiv \widehat{bv}_2 = \begin{cases} \bot_{\widehat{bv}} & \text{if } \widehat{bv}_1 = \bot_{\widehat{bv}} \vee \widehat{bv}_2 = \bot_{\widehat{bv}} \\ \texttt{inject}^\sharp(\textsf{false}) & \text{if } \widehat{bv}_1 \neq \bot_{\widehat{bv}},\ \widehat{bv}_2 \neq \bot_{\widehat{bv}},\ \mathit{types}_1 \cap \mathit{types}_2 = \emptyset \\ \texttt{inject}^\sharp(\mathit{diff?} \cup \mathit{bytype}) & \text{otherwise} \end{cases}$$

where

$$\mathit{types}_1 = \{\text{non-}\bot \text{ components of } \widehat{bv}_1\}$$

$$\mathit{types}_2 = \{\text{non-}\bot \text{ components of } \widehat{bv}_2\}$$

$$\mathit{diff?} = \begin{cases} \emptyset & \text{if } |\mathit{types}_1| = 1,\ \mathit{types}_1 = \mathit{types}_2 \\ \textsf{false} & \text{otherwise} \end{cases}$$

$$\mathit{bytype} = \mathit{num?} \cup \mathit{str?} \cup \mathit{bool?} \cup \mathit{addr?} \cup \mathit{null?} \cup \mathit{undef?}$$

$$\mathit{num?} = \pi_{\hat{n}}(\widehat{bv}_1) \equiv \pi_{\hat{n}}(\widehat{bv}_2)$$

$$\mathit{str?} = \pi_{\widehat{str}}(\widehat{bv}_1) \equiv \pi_{\widehat{str}}(\widehat{bv}_2)$$

$$\mathit{bool?} = \begin{cases} \emptyset & \text{if } \pi_{\hat{b}}(\widehat{bv}_1) \cup \pi_{\hat{b}}(\widehat{bv}_2) = \emptyset \\ \textsf{true} & \text{if } |\pi_{\hat{b}}(\widehat{bv}_1)| = 1,\ \pi_{\hat{b}}(\widehat{bv}_1) = \pi_{\hat{b}}(\widehat{bv}_2) \\ \textsf{false} & \text{if } \pi_{\hat{b}}(\widehat{bv}_1) \cap \pi_{\hat{b}}(\widehat{bv}_2) = \emptyset,\ \pi_{\hat{b}}(\widehat{bv}_1) \cup \pi_{\hat{b}}(\widehat{bv}_2) \neq \emptyset \\ \{\textsf{true}, \textsf{false}\} & \text{otherwise} \end{cases}$$

$$\mathit{addr?} = \begin{cases} \emptyset & \text{if } \pi_{\hat{a}}(\widehat{bv}_1) \cup \pi_{\hat{a}}(\widehat{bv}_2) = \emptyset \\ \textsf{true} & \text{if } \pi_{\hat{a}}(\widehat{bv}_1) = \pi_{\hat{a}}(\widehat{bv}_2) = \{\hat{a}\},\ |\gamma_a(\hat{a})| = 1 \\ \textsf{false} & \text{if } \pi_{\hat{a}}(\widehat{bv}_1) \cap \pi_{\hat{a}}(\widehat{bv}_2) = \emptyset,\ \pi_{\hat{a}}(\widehat{bv}_1) \cup \pi_{\hat{a}}(\widehat{bv}_2) \neq \emptyset \\ \{\textsf{true}, \textsf{false}\} & \text{otherwise} \end{cases}$$

$$\mathit{null?} = \begin{cases} \emptyset & \text{if } \pi_{\widehat{null}}(\widehat{bv}_1) \cup \pi_{\widehat{null}}(\widehat{bv}_2) = \emptyset \\ \textsf{true} & \text{if } \pi_{\widehat{null}}(\widehat{bv}_1) = \textsf{null},\ \pi_{\widehat{null}}(\widehat{bv}_2) = \textsf{null} \\ \textsf{false} & \text{otherwise} \end{cases}$$

$$\mathit{undef?} = \begin{cases} \emptyset & \text{if } \pi_{\widehat{undef}}(\widehat{bv}_1) \cup \pi_{\widehat{undef}}(\widehat{bv}_2) = \emptyset \\ \textsf{true} & \text{if } \pi_{\widehat{undef}}(\widehat{bv}_1) = \textsf{undef},\ \pi_{\widehat{undef}}(\widehat{bv}_2) = \textsf{undef} \\ \textsf{false} & \text{otherwise} \end{cases}$$

where equality of abstract numbers and abstract strings are specific to their given abstract domains.

3.2.5 Non-strict Equality Operator.

For the binary operator $\approx$:

$$\widehat{bv}_1 \approx \widehat{bv}_2 = \begin{cases} \widehat{bv}_1 \equiv \widehat{bv}_2 \sqcup \mathit{case1} \sqcup \mathit{case2} \sqcup \mathit{case3} \sqcup \mathit{case4} & \text{if } \pi_{\hat{b}}(\widehat{bv}_1 \equiv \widehat{bv}_2) = \{\textsf{false}\} \\ \widehat{bv}_1 \equiv \widehat{bv}_2 & \text{otherwise} \end{cases}$$

where

$$\mathit{case1} = \begin{cases} \texttt{inject}^\sharp(\textsf{true}) & \text{if } \pi_{\widehat{null}}(\widehat{bv}_1) = \textsf{null}, \pi_{\widehat{undef}}(\widehat{bv}_2) = \textsf{undef} \\ \emptyset & \text{otherwise} \end{cases}$$

$$\mathit{case2} = \begin{cases} \texttt{inject}^\sharp(\textsf{true}) & \text{if } \pi_{\widehat{undef}}(\widehat{bv}_1) = \textsf{undef}, \pi_{\widehat{null}}(\widehat{bv}_2) = \textsf{null} \\ \emptyset & \text{otherwise} \end{cases}$$

$$\mathit{case3} = \begin{cases} \texttt{inject}^\sharp(\hat{n}) \equiv \textsf{tonum}\ \texttt{inject}^\sharp(\widehat{str}) & \text{if } \pi_{\hat{n}}(\widehat{bv}_1) = \hat{n}, \pi_{\widehat{str}}(\widehat{bv}_2) = \widehat{str} \\ \emptyset & \text{otherwise} \end{cases}$$

$$\mathit{case4} = \begin{cases} \texttt{inject}^\sharp(\hat{n}) \equiv \textsf{tonum}\ \texttt{inject}^\sharp(\widehat{str}) & \text{if } \pi_{\widehat{str}}(\widehat{bv}_1) = \widehat{str}, \pi_{\hat{n}}(\widehat{bv}_2) = \hat{n} \\ \emptyset & \text{otherwise} \end{cases}$$

3.2.6 The Access Operator.

The binary operator . accesses a property of an object. The translator guarantees it is only applied to an address and string.

$$\widehat{bv}_1.\widehat{bv}_2 = \texttt{lookup}^\sharp\left(\pi_{\hat{a}}(\widehat{bv}_1), \pi_{\widehat{str}}(\widehat{bv}_2), \hat{\sigma}\right)$$

3.2.7 The instanceof Operator.

For binary operator **instanceof**:

$$\widehat{bv}_1 \ \textsf{\textbf{instanceof}}\ \widehat{bv}_2 = \texttt{inject}^\sharp\left(\mathit{addr?} \cup \texttt{instance}^\sharp\left(\pi_{\hat{a}}(\widehat{bv}_1), \pi_{\hat{a}}(\widehat{bv}_2)\right)\right) \quad \text{where}$$

$$\mathit{addr?} = \begin{cases} \textsf{\textbf{false}} & \text{if } \widehat{bv}_1 \neq \texttt{inject}^\sharp(\pi_{\hat{a}}(\widehat{bv}_1)) \\ \emptyset & \text{otherwise} \end{cases}$$

$$\texttt{instance}^\sharp(\bar{\hat{a}}_1, \bar{\hat{a}}_2) = \mathit{found} \quad \text{where}$$

$$\widehat{bv} = \bigsqcup_{\hat{a} \in \bar{\hat{a}}_1} \texttt{getProto}^\sharp(\hat{\sigma}(\hat{a}))$$

$$\mathit{proto} = \pi_{\hat{a}}(\widehat{bv})$$

$$\mathit{null?} = \textsf{\textbf{null}} \in \pi_{\widehat{null}}(\widehat{bv})$$

$$\mathit{found} = \begin{cases} \emptyset & \text{if } \bar{\hat{a}}_1 = \emptyset \vee \bar{\hat{a}}_2 = \emptyset \\ \textsf{\textbf{true}} & \text{if } \neg\mathit{null?},\ \mathit{proto} = \bar{\hat{a}}_2 = \{\hat{a}\},\ |\gamma_a(\hat{a})| = 1 \\ \textsf{\textbf{false}} & \text{if } \mathit{null?},\ \mathit{proto} = \emptyset \\ \{\textsf{\textbf{true}}, \textsf{\textbf{false}}\} & \text{if } \mathit{null?},\ \mathit{proto} \cap \bar{\hat{a}}_2 \neq \emptyset \\ \{\textsf{\textbf{true}}\} \cup \texttt{instance}^\sharp(\mathit{proto}, \bar{\hat{a}}_2) & \text{if } \neg\mathit{null?},\ \mathit{proto} \cap \bar{\hat{a}}_2 \neq \emptyset \\ \{\textsf{\textbf{false}}\} \cup \texttt{instance}^\sharp(\mathit{proto}, \bar{\hat{a}}_2) & \text{if } \mathit{null?},\ \mathit{proto} \neq \emptyset,\ \mathit{proto} \cap \bar{\hat{a}}_2 = \emptyset \\ \texttt{instance}^\sharp(\mathit{proto}, \bar{\hat{a}}_2) & \text{otherwise} \end{cases}$$

3.2.8 The in Operator.

For binary operator **in**:

$$\widehat{bv}_1 \ \textsf{\textbf{in}}\ \widehat{bv}_2 = \texttt{inject}^\sharp\left(\bigcup_{\hat{a} \in \pi_{\hat{a}}(\widehat{bv}_2)} \texttt{find}^\sharp(\hat{\sigma}(\hat{a}), \pi_{\widehat{str}}(\widehat{bv}_1), \hat{\sigma})\right) \quad \text{where}$$

$$\texttt{find}^\sharp(\hat{o}, \widehat{str}, \hat{\sigma}) =$$

$$\begin{cases} \textsf{\textbf{true}} & \text{if } \widehat{str} \in \pi_3(\hat{o}) \\ \textsf{\textbf{false}} & \text{if } \nexists \widehat{str}' \in \texttt{dom}(\hat{o}) \,.\, \widehat{str} \sqsubseteq \widehat{str}',\ \pi_{\hat{a}}(\widehat{bv}) = \emptyset \\ \{\textsf{\textbf{true}}, \textsf{\textbf{false}}\} & \text{if } \widehat{str} \notin \pi_3(\hat{o}),\ \exists \widehat{str}' \in \texttt{dom}(\hat{o}) \,.\, \widehat{str} \sqsubseteq \widehat{str}',\ \textsf{\textbf{null}} \in \pi_{\widehat{null}}(\widehat{bv}) \\ \{\textsf{\textbf{true}}\} \cup \mathit{proto} & \text{if } \widehat{str} \notin \pi_3(\hat{o}),\ \exists \widehat{str}' \in \texttt{dom}(\hat{o}) \,.\, \widehat{str} \sqsubseteq \widehat{str}',\ \textsf{\textbf{null}} \notin \pi_{\widehat{null}}(\widehat{bv}) \\ \{\textsf{\textbf{false}}\} \cup \mathit{proto} & \text{if } \nexists \widehat{str}' \in \texttt{dom}(\hat{o}) \,.\, \widehat{str} \sqsubseteq \widehat{str}',\ \pi_{\hat{a}}(\widehat{bv}) \neq \emptyset,\ \textsf{\textbf{null}} \in \pi_{\widehat{null}}(\widehat{bv}) \\ \mathit{proto} & \text{otherwise} \end{cases}$$

where

$$\widehat{bv} = \texttt{getProto}^\sharp(\hat{o})$$

$$\mathit{proto} = \bigcup_{\hat{a} \in \pi_{\hat{a}}(\widehat{bv})} \texttt{find}^\sharp(\hat{\sigma}(\hat{a}), \widehat{str}, \hat{\sigma})$$

3.2.9 Numeric Unary Operators.

For unary operators $\hat{\odot} \in \{-, \sim\}$:

$\hat{\odot}\widehat{bv} = \texttt{inject}^\sharp(\hat{\odot}\pi_{\hat{n}}(\widehat{bv}))$

where the operators in $\hat{\odot}$ are defined on and return $Num^\sharp$ and are specific to a given numeric abstract domain.

3.2.10 Logical Negation Operator.

For unary operator $\neg$:

$$\neg\widehat{bv} = \texttt{inject}^\sharp(\hat{b}) \quad \text{where}$$
$$\hat{b} = \begin{cases} \emptyset & \text{if } \pi_{\hat{b}}(\widehat{bv}) = \emptyset \\ \textsf{true} & \text{if } \pi_{\hat{b}}(\widehat{bv}) = \textsf{false} \\ \textsf{false} & \text{if } \pi_{\hat{b}}(\widehat{bv}) = \textsf{true} \\ \{\textsf{true}, \textsf{false}\} & \text{otherwise} \end{cases}$$

3.2.11 The typeof Operator.

For unary operator **typeof**:

$$\textsf{typeof}\ \widehat{bv} = \texttt{inject}^\sharp\left(\bigsqcup_{i\in[1..7]} \widehat{str}_i\right) \quad \text{where}$$
$$\widehat{str}_1 = \begin{cases} \alpha_{str}(\texttt{"number"}) & \text{if } \pi_{\hat{n}}(\widehat{bv}) \neq \bot_{\hat{n}} \\ \bot_{\widehat{str}} & \text{otherwise} \end{cases}$$
$$\widehat{str}_2 = \begin{cases} \alpha_{str}(\texttt{"boolean"}) & \text{if } \pi_{\hat{b}}(\widehat{bv}) \neq \emptyset \\ \bot_{\widehat{str}} & \text{otherwise} \end{cases}$$
$$\widehat{str}_3 = \begin{cases} \alpha_{str}(\texttt{"string"}) & \text{if } \pi_{\widehat{str}}(\widehat{bv}) \neq \bot_{\widehat{str}} \\ \bot_{\widehat{str}} & \text{otherwise} \end{cases}$$
$$\widehat{str}_4 = \begin{cases} \alpha_{str}(\texttt{"object"}) & \text{if } \pi_{\hat{a}}(\widehat{bv}) \neq \emptyset,\ \exists \hat{a} \in \pi_{\hat{a}}(\widehat{bv})\,.\,\texttt{getClass}^\sharp(\hat{\sigma}(\hat{a})) \neq \textsf{function} \\ \bot_{\widehat{str}} & \text{otherwise} \end{cases}$$
$$\widehat{str}_5 = \begin{cases} \alpha_{str}(\texttt{"function"}) & \text{if } \pi_{\hat{a}}(\widehat{bv}) \neq \emptyset,\ \exists \hat{a} \in \pi_{\hat{a}}(\widehat{bv})\,.\,\texttt{getClass}^\sharp(\hat{\sigma}(\hat{a})) = \textsf{function} \\ \bot_{\widehat{str}} & \text{otherwise} \end{cases}$$
$$\widehat{str}_6 = \begin{cases} \alpha_{str}(\texttt{"object"}) & \text{if } \pi_{\widehat{null}}(\widehat{bv}) \neq \emptyset \\ \bot_{\widehat{str}} & \text{otherwise} \end{cases}$$
$$\widehat{str}_7 = \begin{cases} \alpha_{str}(\texttt{"undefined"}) & \text{if } \pi_{\widehat{undef}}(\widehat{bv}) \neq \emptyset \\ \bot_{\widehat{str}} & \text{otherwise} \end{cases}$$

3.2.12 The tobool Operator.

For unary operator **tobool**:

$$\textsf{\textbf{tobool}}\ \widehat{bv} = \texttt{inject}^\sharp\left(\bigsqcup_{i\in[1..6]} \hat{b}_i\right) \quad \text{where}$$

$$\hat{b}_1 = \begin{cases} \emptyset & \text{if } \overline{n} = \emptyset \\ \textsf{\textbf{false}} & \text{if } \overline{n} \in \{\{\texttt{NaN}\}, \{0\}\} \\ \textsf{\textbf{true}} & \text{if } \texttt{NaN} \notin \overline{n},\ 0 \notin \overline{n} \\ \{\textsf{\textbf{true}}, \textsf{\textbf{false}}\} & \text{otherwise} \end{cases}$$

$$\text{where } \overline{n} = \gamma_n(\pi_{\hat{n}}(\widehat{bv}))$$

$$\hat{b}_2 = \pi_{\hat{b}}(\widehat{bv})$$

$$\hat{b}_3 = \begin{cases} \emptyset & \text{if } \overline{str} = \emptyset \\ \textsf{\textbf{false}} & \text{if } \overline{str} = \{\texttt{""}\} \\ \textsf{\textbf{true}} & \text{if } \{\texttt{""}\} \notin \overline{str} \\ \{\textsf{\textbf{true}}, \textsf{\textbf{false}}\} & \text{otherwise} \end{cases}$$

$$\text{where } \overline{str} = \gamma_{str}(\pi_{\widehat{str}}(\widehat{bv}))$$

$$\hat{b}_4 = \begin{cases} \textsf{\textbf{true}} & \text{if } \pi_{\hat{a}}(\widehat{bv}) \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases}$$

$$\hat{b}_5 = \begin{cases} \textsf{\textbf{false}} & \text{if } \pi_{\widehat{null}}(\widehat{bv}) = \{\textsf{\textbf{null}}\} \\ \emptyset & \text{otherwise} \end{cases}$$

$$\hat{b}_6 = \begin{cases} \textsf{\textbf{false}} & \text{if } \pi_{\widehat{undef}}(\widehat{bv}) = \{\textsf{\textbf{undef}}\} \\ \emptyset & \text{otherwise} \end{cases}$$

3.2.13 The isprim Operator.

The unary operator **isprim** determines whether a value is primitive or references an object.

$$\textsf{\textbf{isprim}}\ \widehat{bv} = \texttt{inject}^\sharp(\hat{b}) \quad \text{where}$$

$$\hat{b} = \begin{cases} \emptyset & \text{if } \widehat{bv} = \bot_{\widehat{bv}} \\ \textsf{\textbf{true}} & \text{if } \widehat{bv} \neq \bot_{\widehat{bv}},\ \pi_{\hat{a}}(\widehat{bv}) = \emptyset \\ \textsf{\textbf{false}} & \text{if } \widehat{bv} \neq \bot_{\widehat{bv}},\ \texttt{inject}^\sharp(\pi_{\hat{a}}(\widehat{bv})) = \widehat{bv} \\ \{\textsf{\textbf{true}}, \textsf{\textbf{false}}\} & \text{otherwise} \end{cases}$$

3.2.14 The tostr Operator.

The **tostr** unary operator converts a primitive value to a string. The translator guarantees that it is never called on an address.

$\textsf{\textbf{tostr}}\ \widehat{bv} = \texttt{inject}^\sharp(num? \sqcup true? \sqcup false? \sqcup str? \sqcup null? \sqcup undef?)$ where

$$num? = \pi_{\hat{n}}(\widehat{bv}).toString$$

$$true? = \begin{cases} \alpha_{str}(\texttt{"true"}) & \text{if } \textsf{\textbf{true}} \in \pi_{\hat{b}}(\widehat{bv}) \\ \bot_{\widehat{str}} & \text{otherwise} \end{cases}$$

$$false? = \begin{cases} \alpha_{str}(\texttt{"false"}) & \text{if } \textsf{\textbf{false}} \in \pi_{\hat{b}}(\widehat{bv}) \\ \bot_{\widehat{str}} & \text{otherwise} \end{cases}$$

$$str? = \pi_{\widehat{str}}(\widehat{bv})$$

$$null? = \begin{cases} \alpha_{str}(\texttt{"null"}) & \text{if } \textsf{\textbf{undef}} \in \pi_{\widehat{null}}(\widehat{bv}) \\ \bot_{\widehat{str}} & \text{otherwise} \end{cases}$$

$$undef? = \begin{cases} \alpha_{str}(\texttt{"undefined"}) & \text{if } \textsf{\textbf{undef}} \in \pi_{\widehat{undef}}(\widehat{bv}) \\ \bot_{\widehat{str}} & \text{otherwise} \end{cases}$$

where the *toString* operation on abstract numbers is specific to a given abstract numeric domain.

3.2.15 The tonum Operator.

The **tonum** unary operator converts a primitive value to a number. The translator guarantees that it is never called on an address.

$\textsf{\textbf{tonum}}\ \widehat{bv} = \texttt{inject}^\sharp(num? \sqcup true? \sqcup false? \sqcup str? \sqcup null? \sqcup undef?)$ where

$$num? = \pi_{\hat{n}}(\widehat{bv})$$

$$true? = \begin{cases} \alpha_n(1) & \text{if } \textsf{\textbf{true}} \in \pi_{\hat{b}}(\widehat{bv}) \\ \bot_{\hat{n}} & \text{otherwise} \end{cases}$$

$$false? = \begin{cases} \alpha_n(0) & \text{if } \textsf{\textbf{false}} \in \pi_{\hat{b}}(\widehat{bv}) \\ \bot_{\hat{n}} & \text{otherwise} \end{cases}$$

$$str? = \pi_{\widehat{str}}(\widehat{bv}).toDouble$$

$$null? = \begin{cases} \alpha_n(0) & \text{if } \textsf{\textbf{undef}} \in \pi_{\widehat{null}}(\widehat{bv}) \\ \bot_{\hat{n}} & \text{otherwise} \end{cases}$$

$$undef? = \begin{cases} \alpha_n(\texttt{NaN}) & \text{if } \textsf{\textbf{undef}} \in \pi_{\widehat{undef}}(\widehat{bv}) \\ \bot_{\hat{n}} & \text{otherwise} \end{cases}$$

where the *toDouble* operation on abstract strings is specific to a given abstract string domain.

3.3 Abstract State Transition Rules

In this section (and this section *only*) we use the notation $\hat{\sigma}\left[\bar{\hat{a}} \mapsto \widehat{bv}\right]$ as shorthand for $\texttt{update}^\sharp(\hat{\sigma}, \bar{\hat{a}}, \widehat{bv})$.

Table 2: The abstract semantics transition relation $\mathcal{F}^\sharp$. Each rule is describes a transition relation from one abstract state $\langle \hat{t}, \hat{\rho}, \hat{\sigma}, \hat{\kappa} \rangle$ to the next abstract state $\langle \hat{t}_{new}, \hat{\rho}_{new}, \hat{\sigma}_{new}, \hat{\kappa}_{new} \rangle$. We use :: to indicate concatenation of sequences and _ to indicate "don't care".

#	$\hat{t}$	Premises	$\hat{t}_{new}$	$\hat{\rho}_{new}$	$\hat{\sigma}_{new}$	$\hat{\kappa}_{new}$
1	$\textsf{\textbf{decl}}\ \overrightarrow{x_i = e_i}\ \textsf{\textbf{in}}\ s$	$\overrightarrow{\widehat{bv}_i} = [\![\vec{e_i}]\!], (\hat{\sigma}', \vec{\hat{a}}) = \texttt{alloc}^\sharp(\hat{\sigma}, \overrightarrow{\widehat{bv}_i}), \hat{\rho}' = \hat{\rho}[\overrightarrow{x_i \mapsto \hat{a}_i}]$	s	$\hat{\rho}'$	$\hat{\sigma}'$	κ
2	$s :: \vec{s_i}$		s	$\hat{\rho}$	$\hat{\sigma}$	$\widehat{\textsf{\textbf{seqK}}}\ \vec{s_i}\ \hat{\kappa}$
3	$\textsf{\textbf{if}}\ e\ s_1\ s_2$	$\textsf{\textbf{true}} \in \pi_{\hat{b}}([\![e]\!])$	s_1	$\hat{\rho}$	$\hat{\sigma}$	$\hat{\kappa}$
4	$\textsf{\textbf{if}}\ e\ s_1\ s_2$	$\textsf{\textbf{false}} \in \pi_{\hat{b}}([\![e]\!])$	s_2	$\hat{\rho}$	$\hat{\sigma}$	$\hat{\kappa}$
5	$x := e$	$\widehat{bv} = [\![e]\!]$	$\widehat{bv}$	$\hat{\rho}$	$\hat{\sigma}[\hat{\rho}(x) \mapsto \widehat{bv}]$	$\hat{\kappa}$
6	$\textsf{\textbf{while}}\ e\ s$	$\textsf{\textbf{true}} \in \pi_{\hat{b}}([\![e]\!])$	s	$\hat{\rho}$	$\hat{\sigma}$	$\widehat{\textsf{\textbf{whileK}}}\ e\ s\ \hat{\kappa}$
7	$\textsf{\textbf{while}}\ e\ s$	$\textsf{\textbf{false}} \in \pi_{\hat{b}}([\![e]\!])$	$\texttt{inject}^\sharp(\textsf{\textbf{undef}})$	$\hat{\rho}$	$\hat{\sigma}$	$\hat{\kappa}$
8	$x := \textsf{\textbf{newfun}}\ m\ n$	$(\widehat{bv}, \hat{\sigma}') = \texttt{allocFun}^\sharp((\hat{\rho}, m), [\![n]\!], \hat{\sigma})$	$\widehat{bv}$	$\hat{\rho}$	$\hat{\sigma}'[\hat{\rho}(x) \mapsto \widehat{bv}]$	$\hat{\kappa}$
9	$x := \textsf{\textbf{new}}\ e_1(e_2)$	$(\hat{\sigma}', \widehat{bv}) = \texttt{allocObj}^\sharp([\![e_1]\!], \hat{\sigma}),$	$\pi_t(\hat{\varsigma})$	$\pi_{\hat{\rho}}(\hat{\varsigma})$	$\pi_{\hat{\sigma}}(\hat{\varsigma})$	$\pi_{\hat{\kappa}}(\hat{\varsigma})$
		$\hat{\sigma}'' = \texttt{setConstr}^\sharp(\hat{\sigma}'[\hat{\rho}(x) \mapsto \widehat{bv}], [\![e_2]\!]),$				
		$\hat{\varsigma} \in \texttt{applyClo}^\sharp([\![e_1]\!], \widehat{bv}, [\![e_2]\!], x, \hat{\rho}, \hat{\sigma}'', \hat{\kappa}),$				
10	$x := \textsf{\textbf{toobj}}\ e$	$((\widehat{bv}, \hat{\sigma}'), \_) = \texttt{toObj}^\sharp([\![e]\!], x, \hat{\rho}, \hat{\sigma})$	$\widehat{bv}$	$\hat{\rho}$	$\hat{\sigma}'$	$\hat{\kappa}$
11	$x := \textsf{\textbf{toobj}}\ e$	$(\_, \widehat{ev}) = \texttt{toObj}^\sharp([\![e]\!], x, \hat{\rho}, \hat{\sigma})$	$\widehat{ev}$	$\hat{\rho}$	$\hat{\sigma}$	$\hat{\kappa}$
12	$e_1.e_2 := e_3$	$((\widehat{bv}, \hat{\sigma}'), \_) = \texttt{updateObj}^\sharp([\![e_1]\!], [\![e_2]\!], [\![e_3]\!], \hat{\sigma})$	$\widehat{bv}$	$\hat{\rho}$	$\hat{\sigma}'$	$\hat{\kappa}$
13	$e_1.e_2 := e_3$	$(\_, \widehat{ev}) = \texttt{updateObj}^\sharp([\![e_1]\!], [\![e_2]\!], [\![e_3]\!], \hat{\sigma})$	$\widehat{ev}$	$\hat{\rho}$	$\hat{\sigma}$	$\hat{\kappa}$
14	$x := \textsf{\textbf{del}}\ e_1.e_2$	$(\hat{\sigma}', \_) = \texttt{delete}^\sharp([\![e_1]\!], [\![e_2]\!], x, \hat{\rho}, \hat{\sigma})$	$\texttt{inject}^\sharp(\textsf{\textbf{undef}})$	$\hat{\rho}$	$\hat{\sigma}'$	$\hat{\kappa}$
15	$x := \textsf{\textbf{del}}\ e_1.e_2$	$(\_, \widehat{ev}) = \texttt{delete}^\sharp([\![e_1]\!], [\![e_2]\!], x, \hat{\rho}, \hat{\sigma})$	$\widehat{ev}$	$\hat{\rho}$	$\hat{\sigma}$	$\hat{\kappa}$
16	$\textsf{\textbf{try-catch-fin}}\ s_1\ x\ s_2\ s_3$		s_1	$\hat{\rho}$	$\hat{\sigma}$	$\widehat{\textsf{\textbf{tryK}}}\ x\ s_2\ s_3\ \hat{\kappa}$
17	$\textsf{\textbf{throw}}\ e$	$\widehat{bv} = [\![e]\!]$	$\textsf{\textbf{exc}}\ \widehat{bv}$	$\hat{\rho}$	$\hat{\sigma}$	$\hat{\kappa}$
18	$\textsf{\textbf{jump}}\ \ell\ e$	$\widehat{bv} = [\![e]\!]$	$\textsf{\textbf{jmp}}\ \ell\ \widehat{bv}$	$\hat{\rho}$	$\hat{\sigma}$	$\hat{\kappa}$
19	$\ell\ s$		s	$\hat{\rho}$	$\hat{\sigma}$	$\widehat{\textsf{\textbf{lblK}}}\ \ell\ \hat{\kappa}$
20	$x := e_1(e_2, e_3)$	$\hat{\varsigma} \in \texttt{applyClo}^\sharp([\![e_1]\!], [\![e_2]\!], [\![e_3]\!], x, \hat{\rho}, \hat{\sigma}, \hat{\kappa}),$	$\pi_t(\hat{\varsigma})$	$\pi_{\hat{\rho}}(\hat{\varsigma})$	$\pi_{\hat{\sigma}}(\hat{\varsigma})$	$\pi_{\hat{\kappa}}(\hat{\varsigma})$
21	$\textsf{\textbf{for}}\ x\ e\ s$	$\overline{\widehat{str}} = \texttt{objKeys}^\sharp([\![e]\!], \hat{\sigma}),\ \widehat{str} \in \overline{\widehat{str}},\ \widehat{bv} = \texttt{inject}^\sharp(\widehat{str})$	s	$\hat{\rho}$	$\hat{\sigma}[\hat{\rho}(x) \mapsto \widehat{bv}]$	$\widehat{\textsf{\textbf{forK}}}\ \widehat{bv}\ x\ s\ \hat{\kappa}$
22	$\textsf{\textbf{for}}\ x\ e\ s$	$\emptyset = \texttt{objKeys}^\sharp([\![e]\!], \hat{\sigma})$	$\texttt{inject}^\sharp(\textsf{\textbf{undef}})$	$\hat{\rho}$	$\hat{\sigma}$	$\hat{\kappa}$
23	$\widehat{bv}$	$\hat{\kappa} = \widehat{\textsf{\textbf{seqK}}}\ s :: \vec{s_i}\ \hat{\kappa}_c$	s	$\hat{\rho}$	$\hat{\sigma}$	$\widehat{\textsf{\textbf{seqK}}}\ \vec{s_i}\ \hat{\kappa}_c$
24	$\widehat{bv}$	$\hat{\kappa} = \widehat{\textsf{\textbf{seqK}}}\ \emptyset\ \hat{\kappa}_c$	$\widehat{bv}$	$\hat{\rho}$	$\hat{\sigma}$	$\hat{\kappa}_c$
25	$\widehat{bv}$	$\hat{\kappa} = \widehat{\textsf{\textbf{whileK}}}\ e\ s\ \hat{\kappa}_c,\ \textsf{\textbf{true}} \in \pi_{\hat{b}}([\![e]\!])$	s	$\hat{\rho}$	$\hat{\sigma}$	$\hat{\kappa}$
26	$\widehat{bv}$	$\hat{\kappa} = \widehat{\textsf{\textbf{whileK}}}\ e\ s\ \hat{\kappa}_c,\ \textsf{\textbf{false}} \in \pi_{\hat{b}}([\![e]\!])$	$\texttt{inject}^\sharp(\textsf{\textbf{undef}})$	$\hat{\rho}$	$\hat{\sigma}$	$\hat{\kappa}_c$
27	$\widehat{bv}$	$\kappa = \widehat{\textsf{\textbf{forK}}}\ \widehat{bv}\ x\ s\ \kappa_c$	s	$\hat{\rho}$	$\hat{\sigma}[\hat{\rho}(x) \mapsto \widehat{bv}]$	$\hat{\kappa}$
28	$\widehat{bv}$	$\kappa = \widehat{\textsf{\textbf{forK}}}\ \widehat{bv}\ x\ s\ \kappa_c$	$\texttt{inject}^\sharp(\textsf{\textbf{undef}})$	$\hat{\rho}$	$\hat{\sigma}$	κ_c
29	$\hat{v}$	$\hat{\kappa} = \widehat{\textsf{\textbf{addrK}}}\ \hat{a},\ \hat{\kappa}' \in \hat{\sigma}(\hat{a})$	$\hat{v}$	$\hat{\rho}$	$\hat{\sigma}$	$\hat{\kappa}'$
30	$\widehat{bv}$	$\hat{\kappa} = \widehat{\textsf{\textbf{retK}}}\ x\ \hat{\rho}_c\ \hat{\kappa}_c\ \textsf{\textbf{call}}$	$\widehat{bv}$	$\hat{\rho}_c$	$\hat{\sigma}[\hat{\rho}_c(x) \mapsto \widehat{bv}]$	$\hat{\kappa}_c$
31	$\widehat{bv}$	$\hat{\kappa} = \widehat{\textsf{\textbf{retK}}}\ x\ \hat{\rho}_c\ \hat{\kappa}_c\ \textsf{\textbf{ctor}},\ \overline{\hat{a}} = \pi_{\hat{a}}(\widehat{bv}),\ \lvert\overline{\hat{a}}\rvert > 0,\ \widehat{bv}' = \texttt{inject}^\sharp(\overline{\hat{a}})$	$\widehat{bv}'$	$\hat{\rho}_c$	$\hat{\sigma}[\hat{\rho}_c(x) \mapsto \widehat{bv}']$	$\hat{\kappa}_c$
32	$\widehat{bv}$	$\hat{\kappa} = \widehat{\textsf{\textbf{retK}}}\ x\ \hat{\rho}_c\ \hat{\kappa}_c\ \textsf{\textbf{ctor}},\ \widehat{bv} \neq \texttt{inject}^\sharp(\pi_{\hat{a}}(\widehat{bv}))$	$[\![x]\!]$	$\hat{\rho}_c$	$\hat{\sigma}$	$\hat{\kappa}_c$
33	$\widehat{ev}$	$\hat{\kappa} = \widehat{\textsf{\textbf{retK}}}\ x\ \hat{\rho}_c\ \hat{\kappa}_c\ \_$	$\widehat{ev}$	$\hat{\rho}_c$	$\hat{\sigma}$	$\hat{\kappa}_c$
34	$\widehat{bv}$	$\hat{\kappa} = \widehat{\textsf{\textbf{tryK}}}\ x\ s_c\ s_f\ \hat{\kappa}_c$	s_f	$\hat{\rho}$	$\hat{\sigma}$	$\widehat{\textsf{\textbf{finK}}}\ \texttt{inject}^\sharp(\textsf{\textbf{undef}})\ \hat{\kappa}_c$

35	$\widehat{jv}$	$\hat{\kappa} = \widehat{\textsf{\textbf{tryK}}}\ x\, s_c\, s_f\, \hat{\kappa}_c$	s_f	$\hat{\rho}$	$\hat{\sigma}$	$\widehat{\textsf{\textbf{finK}}}\ \widehat{jv}\, \hat{\kappa}_c$
36	$\textsf{\textbf{exc}}\ \widehat{bv}$	$\hat{\kappa} = \widehat{\textsf{\textbf{tryK}}}\ x\, s_c\, s_f\, \hat{\kappa}_c$	s_c	$\hat{\rho}$	$\hat{\sigma}[\hat{\rho}(x) \mapsto \widehat{bv}]$	$\widehat{\textsf{\textbf{catchK}}}\ s_f\, \hat{\kappa}_c$
37	$\widehat{bv}$	$\hat{\kappa} = \widehat{\textsf{\textbf{catchK}}}\ s_f\, \hat{\kappa}_c$	s_f	$\hat{\rho}$	$\hat{\sigma}$	$\widehat{\textsf{\textbf{finK}}}\ \texttt{inject}^\sharp(\textsf{\textbf{undef}})\, \hat{\kappa}_c$
38	$\hat{v}$	$\hat{\kappa} = \widehat{\textsf{\textbf{catchK}}}\ s_f\, \hat{\kappa}_c,\ \hat{v} \in EValue^\sharp \cup JValue^\sharp$	s_f	$\hat{\rho}$	$\hat{\sigma}$	$\widehat{\textsf{\textbf{finK}}}\ \hat{v}\, \hat{\kappa}_c$
39	$\widehat{bv}$	$\hat{\kappa} = \widehat{\textsf{\textbf{finK}}}\ \overline{\hat{v}}\, \hat{\kappa}_c,\ \hat{v} \in \overline{\hat{v}},\ \hat{v}' = \hat{v} \in EValue^\sharp \cup JValue^\sharp\ ?\ \hat{v} : \widehat{bv}$	$\hat{v}'$	$\hat{\rho}$	$\hat{\sigma}$	$\hat{\kappa}_c$
40	$\hat{v}$	$\hat{\kappa} = \textsf{\textbf{lblK}}\ \ell\, \hat{\kappa}_c,\ (\hat{v} = \textsf{\textbf{jmp}}\ \ell\ \widehat{bv} \vee \hat{v} = \widehat{bv})$	$\widehat{bv}$	$\hat{\rho}$	$\hat{\sigma}$	$\hat{\kappa}_c$
41	$\textsf{\textbf{jmp}}\ \ell\ \widehat{bv}$	$\hat{\kappa} = \widehat{\textsf{\textbf{lblK}}}\ \ell'\, \hat{\kappa}_c,\ \ell \neq \ell'$	$\textsf{\textbf{jmp}}\ \ell\ \widehat{bv}$	$\hat{\rho}$	$\hat{\sigma}$	$\hat{\kappa}_c$
42	$\hat{v}$	$\hat{\kappa} \notin \texttt{specialK}^\sharp(\hat{v}),\ \hat{v} \in EValue^\sharp \cup JValue^\sharp$	$\hat{v}$	$\hat{\rho}$	$\hat{\sigma}$	$\texttt{nextK}^\sharp(\hat{\kappa})$

3.4 Abstract Helper Functions

We define the abstract helper functions used by the previous sections. The functions are listed in alphabetical order. In this section, unlike the previous section, the notation $\hat{\sigma}[\hat{a} \mapsto \_]$ is used for a direct update to the abstract store, *not* as shorthand for a call to $\texttt{update}^\sharp$. Again we use the notation $\mathcal{O}(\cdot)$ to indicate an "Option" type: essentially a set guaranteed to contain zero or one values.

3.4.1 $\texttt{alloc}^\sharp$

$\texttt{alloc}^\sharp$ takes a store and a list of base values (or a continuation) and returns a new store and a list of addresses such that the new store maps the addresses to the given values. If the addresses already exist in the store, which can happen since there are a finite number of abstract addresses allowed, then the given arguments are joined with the existing values at those addresses. Note that the heap sensitivity strategy should guarantee that base values $\widehat{bv} \in BValue^\sharp$, objects $\hat{o} \in Object^\sharp$ (handled in $\texttt{allocObj}^\sharp$), and continuations $\hat{\kappa} \in Kont^\sharp$ are always given addresses from disjoint sets; under this assumption the semantics guarantees that any given address will always map to only one of those three domains. We assume that either all the addresses are fresh or all of them are already in the store.

$$\texttt{alloc}^\sharp \in (Store^\sharp \times \overrightarrow{BValue^\sharp}_i \to Store^\sharp \times \overrightarrow{Address^\sharp}_i) + (Store^\sharp \times Kont^\sharp \to Store^\sharp \times Address^\sharp)$$

$$\texttt{alloc}^\sharp(\hat{\sigma}, \overrightarrow{\widehat{bv}_i}) = (\hat{\sigma}', \vec{\hat{a}}) \quad \text{where}$$

$\quad \vec{\hat{a}}$ depends on the heap sensitivity strategy

$$\hat{\sigma} = \begin{cases} \hat{\sigma}[\overrightarrow{\hat{a}_i \mapsto \widehat{bv}_i}] & \text{if } \vec{\hat{a}} \cap \texttt{dom}(\hat{\sigma}) = \emptyset \\ \hat{\sigma}[\overrightarrow{\hat{a}_i \mapsto \hat{\sigma}(\hat{a}_i) \sqcup \widehat{bv}_i}] & \text{if } \vec{\hat{a}} \subseteq \texttt{dom}(\hat{\sigma}) \end{cases}$$

$$\texttt{alloc}^\sharp(\hat{\sigma}, \hat{\kappa}) = (\hat{\sigma}', \hat{a}) \quad \text{where}$$

$\quad \hat{a}$ depends on the heap sensitivity strategy

$$\hat{\sigma} = \begin{cases} \hat{\sigma}[\hat{a} \mapsto \{\hat{\kappa}\}] & \text{if } \hat{a} \notin \texttt{dom}(\hat{\sigma}) \\ \hat{\sigma}[\hat{a} \mapsto \hat{\sigma}(\hat{a}) \cup \{\hat{\kappa}\}] & \text{if } \hat{a} \in \texttt{dom}(\hat{\sigma}) \end{cases}$$

3.4.2 $\texttt{allocFun}^\sharp$

$\texttt{allocFun}$ is used to allocate a function object into the abstract store, setting the properties appropriately.

$$\texttt{allocFun}^\sharp \in Closure^\sharp \times BValue^\sharp \times Store^\sharp \to Store^\sharp \times BValue^\sharp$$

$$\texttt{allocFun}^\sharp(\widehat{clo}, \widehat{bv}, \hat{\sigma}) = \big(\hat{\sigma}', \texttt{inject}^\sharp(\hat{a})\big) \quad \text{where}$$

$\quad \hat{a}$ depends on the heap sensitivity strategy

$$internal = [\texttt{"proto"} \mapsto \texttt{inject}^\sharp(\widehat{\texttt{Function\_prototype\_Addr}}), \texttt{"class"} \mapsto \textbf{function}, \texttt{"code"} \mapsto \widehat{clo}]$$

$$external = [\alpha_{str}(\texttt{"length"}) \mapsto \widehat{bv}]$$

$$\hat{o} = (external, internal, \{\alpha_{str}(\texttt{"length"})\})$$

$$\hat{\sigma}' = \begin{cases} \hat{\sigma}[\hat{a} \mapsto \hat{o}] & \text{if } \hat{a} \notin \texttt{dom}(\hat{\sigma}) \\ \hat{\sigma}[\hat{a} \mapsto \hat{o} \sqcup \hat{\sigma}(\hat{a})] & \text{otherwise} \end{cases}$$

3.4.3 $\texttt{allocObj}^\sharp$

$\texttt{allocObj}^\sharp$ allocates objects into the store; it also initializes them suitably based on the first argument passed in, which is the address referring to the object's constructor. If there is an object already at the allocation address (selected by the heap sensitivity strategy) then the new object is joined with the old object. Note that the heap sensitivity strategy should guarantee that the sets of addresses given to objects from different classes are always disjoint. For each of the object classes that the constructor passed in as the first argument can create, the corresponding `"prototype"` is extracted and the objects created in that class have their `"proto"` set according to that value. The mapping from special addresses to classes is defined in `builtin.pdf`; the helper function `classFromAddress` is suitably abstracted for use here. Recall that $c \in Class$; the notation $\overline{X}\lfloor_c$ indicates the set $\overline{X}$ has one element for each class c and we're accessing the c^{th} element of the set.

$$\texttt{allocObj}^\sharp \in \mathit{BValue}^\sharp \times \mathit{Store}^\sharp \to \mathit{Store}^\sharp \times \mathit{BValue}^\sharp$$

$$\texttt{allocObj}^\sharp(\widehat{bv}, \hat{\sigma}) = \big(\hat{\sigma}', \texttt{inject}^\sharp(\bar{\hat{a}})\big) \quad \text{where}$$

$$\overline{addrs}\rfloor_c = \left\{\ \hat{a} \in \pi_{\hat{a}}(\widehat{bv})\ \mid\ \texttt{classFromAddress}(\hat{a}) = c\ \right\}$$

$\bar{\hat{a}}\rfloor_c$ depends on the heap sensitivity strategy

$$\overline{\hat{a}'}\rfloor_c = \big(\{\hat{a}'\ \mid\ \hat{a} \in \overline{addrs}\rfloor_c,\ \hat{a}' \in \pi_{\hat{a}}(\hat{\sigma}(\hat{a})(\texttt{"prototype"}))\}\big) \bigsqcup \begin{cases} \emptyset & \text{if } \forall \hat{a} \in \overline{addrs}\rfloor_c\ .\texttt{inject}^\sharp(\pi_{\hat{a}}(\hat{\sigma}(\hat{a})(\texttt{"prototype"}))) = \hat{\sigma}(\hat{a})(\texttt{"prototype"}) \\ \widehat{\texttt{Object\_prototype\_Addr}} & \text{otherwise} \end{cases}$$

$$\overline{internal}\rfloor_c = \begin{cases} [\texttt{"proto"} \mapsto \texttt{inject}^\sharp(\overline{\hat{a}'}\rfloor_c), \texttt{"class"} \mapsto c] & \text{if } |\overline{addrs}\rfloor_c| > 0 \\ \emptyset & \text{otherwise} \end{cases}$$

$$\bar{\hat{o}}\rfloor_c = (\emptyset, \overline{internal}\rfloor_c, \emptyset)$$

$$\bar{\hat{\sigma}}\rfloor_c = \begin{cases} \hat{\sigma}[\overrightarrow{\bar{\hat{a}}\rfloor_c \mapsto \bar{\hat{o}}\rfloor_c}] & \text{if } \bar{\hat{a}}\rfloor_c \notin \texttt{dom}(\hat{\sigma}) \\ \hat{\sigma}[\overrightarrow{\bar{\hat{a}}\rfloor_c \mapsto \bar{\hat{o}}\rfloor_c \sqcup \hat{\sigma}(\bar{\hat{a}}\rfloor_c)}] & \text{otherwise} \end{cases}$$

$$\hat{\sigma}' = \bigsqcup \left\{\ \bar{\hat{\sigma}}\rfloor_c\ \mid\ |\overline{addrs}\rfloor_c| > 0\ \right\}$$

3.4.4 `applyClo`$^\sharp$

`applyClo`$^\sharp$ is used to make method calls. It returns a set of states containing the callee states if the call can succeed given a set of closures and also an exception state if the call may be to a non-`callable` object. The predicate `callable` maps those classes that have an internal `"code"` field to **true** and the rest of the classes to **false**. For convenience we use *body* to refer to either a statement $s \in \mathit{Stmt}$ or a declaration $d \in \mathit{Decl}$, since the callee's body can be either one and we do the same thing in either case.

$$\texttt{applyClo}^\sharp \in \mathit{BValue}^\sharp \times \mathit{BValue}^\sharp \times \mathit{BValue}^\sharp \times \mathit{Variable} \times \mathit{Env}^\sharp \times \mathit{Store}^\sharp \times \mathit{Kont}^\sharp \to \mathcal{P}(\mathit{State}^\sharp)$$

$$\texttt{applyClo}^\sharp(\widehat{bv}_1, \widehat{bv}_2, \widehat{bv}_3, x, \hat{\rho}, \hat{\sigma}, \hat{\kappa}) = \bar{\hat{\varsigma}}\ \cup\ \hat{\varsigma}_{exc} \quad \text{where}$$

$$\widehat{bv}_4 = \texttt{inject}^\sharp(\pi_{\hat{a}}(\widehat{bv}_2))$$

$$\widehat{bv}_5 = \texttt{inject}^\sharp(\pi_{\hat{a}}(\widehat{bv}_3))$$

$$\bar{\hat{a}} = \left\{\hat{a} \in \pi_{\hat{a}}(\widehat{bv})\ \mid\ \texttt{callable}(\texttt{getClass}^\sharp(\hat{\sigma}(\hat{a}))) = \textbf{true}\right\}$$

$$\begin{aligned} \bar{\hat{\varsigma}} = \{\ (\mathit{body}, \hat{\rho}_c', \hat{\sigma}'', \widehat{\textbf{addrK}}\ \hat{a}')\ \mid\ & \widehat{bv}_4 \neq \bot_{\widehat{bv}},\ \widehat{bv}_5 \neq \bot_{\widehat{bv}},\ \hat{a} \in \bar{\hat{a}} \\ & (\hat{\rho}_c, (\texttt{self}, \texttt{args}) \Rightarrow \mathit{body}) \in \pi_2(\hat{\sigma}(\hat{a}))(\texttt{"code"}), \\ & (\hat{\sigma}', \{\hat{a}_2', \hat{a}_3'\}) = \texttt{alloc}^\sharp(\hat{\sigma}, \{\widehat{bv}_4, \widehat{bv}_5\}), \\ & \hat{\rho}_c' = \hat{\rho}_c[\texttt{self} \mapsto \hat{a}_2',\ \texttt{args} \mapsto \hat{a}_3'], \\ & \mathit{ctxt} = \begin{cases} \textbf{ctor} & \text{if } \{\hat{a}_3\} = \pi_{\hat{a}}(\widehat{bv}_3), \texttt{"constructor"} \in \texttt{dom}(\pi_2(\hat{\sigma}'(\hat{a}_3))) \\ \textbf{call} & \text{otherwise} \end{cases}, \\ & (\hat{\sigma}'', \hat{a}') = \texttt{alloc}^\sharp(\hat{\sigma}', \widehat{\textbf{retK}}\ x\ \hat{\rho}\ \hat{\kappa}\ \mathit{ctxt})\ \} \end{aligned}$$

$$\hat{\varsigma}_{exc} = \begin{cases} (\textbf{exc}\ \alpha_{str}(\texttt{"TypeError"}), \hat{\rho}, \hat{\sigma}, \hat{\kappa}) & \text{if } \widehat{bv}_1 \neq \texttt{inject}^\sharp(\pi_{\hat{a}}(\widehat{bv}_1)) \vee |\bar{\hat{a}}| = 0 \vee \bar{\hat{a}} \neq \pi_{\hat{a}}(\widehat{bv}_1) \vee \widehat{bv}_4 = \bot_{\widehat{bv}} \vee \widehat{bv}_5 = \bot_{\widehat{bv}} \\ \emptyset & \text{otherwise} \end{cases}$$

3.4.5 `delete`$^\sharp$

`delete`$^\sharp$ removes a property from an abstract object. `delete`$^\sharp$ returns a tuple; the first component is the new store when `delete`$^\sharp$ does not raise an exception, and the second component is for the case when it does. `delete`$^\sharp$ does not delete any property whose attributes include **nodelete**. The subtraction operation $\hat{o} - \widehat{str}$ removes the given abstract string from $\hat{o}$'s programmer-visible map *and* set of definitely-present properties.

$\texttt{delete}^\sharp \in BValue^\sharp \times BValue^\sharp \times Variable \times Env^\sharp \times Store^\sharp \rightarrow \mathcal{O}(Store^\sharp) \times \mathcal{O}(EValue^\sharp)$

$\texttt{delete}^\sharp(\widehat{bv}_1, \widehat{bv}_2, x, \hat{\rho}, \hat{\sigma}) = (noexc, exc) \quad \text{where}$

$$\widehat{str} = \pi_{\widehat{str}}(\widehat{bv}_2)$$

$$objs = \left\{ (\hat{a}, \hat{o}) \mid \hat{a} \in \pi_{\hat{a}}(\widehat{bv}_1),\ \hat{o} = \hat{\sigma}(\hat{a}) \right\}$$

$$strong? = objs = \{(\hat{a}, \_)\},\ |\gamma_a(\hat{a})| = 1$$

$$defPresent? = \forall \hat{o} \in objs\,.\, \widehat{str} \in \pi_3(\hat{o})$$

$$defAbsent? = \forall \hat{o} \in objs, \forall \widehat{str}' \in \texttt{dom}(\hat{o})\,.\, \widehat{str} \not\sqsubseteq \widehat{str}'$$

$$newobjs = \left\{ (\hat{a}, \hat{o} - \widehat{str}) \mid (\hat{a}, \hat{o}) \in objs \right\}$$

$$\hat{\sigma}_t = \texttt{update}^\sharp(\hat{\sigma}, \hat{\rho}(x), \texttt{inject}^\sharp(\textbf{true}))$$

$$\hat{\sigma}_f = \texttt{update}^\sharp(\hat{\sigma}, \hat{\rho}(x), \texttt{inject}^\sharp(\textbf{false}))$$

$$\hat{\sigma}_\top = \texttt{update}^\sharp(\hat{\sigma}, \hat{\rho}(x), \texttt{inject}^\sharp(\{\textbf{true}, \textbf{false}\}))$$

$$noexc = \begin{cases} \hat{\sigma}_f & \text{if } defAbsent? \\ \hat{\sigma}_t[\hat{a} \mapsto \hat{o}] & \text{if } defPresent?,\ strong? \\ \hat{\sigma}_t[\overrightarrow{\hat{a}_i \mapsto \hat{\sigma}(\hat{a}_i) \sqcup \hat{o}_i}] & \text{if } defPresent?,\ \neg strong?,\ (\hat{a}_i, \hat{o}_i) \in newobjs \\ \hat{\sigma}_\top[\overrightarrow{\hat{a}_i \mapsto \hat{\sigma}(\hat{a}_i) \sqcup \hat{o}_i}] & \text{if } \neg defAbsent,\ \neg defPresent,\ |newobjs| > 0,\ (\hat{a}_i, \hat{o}_i) \in newobjs \\ \emptyset & \text{otherwise} \end{cases}$$

$$exc = \begin{cases} \textbf{exc}\ \alpha_{str}(\texttt{"TypeError"}) & \text{if } \pi_{\widehat{null}}(\widehat{bv}_1) \bigcup \pi_{\widehat{undef}}(\widehat{bv}_1) \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases}$$

3.4.6 $\texttt{getProto}^\sharp$

$\texttt{getProto}^\sharp$ retrieves the `"proto"` property from an object's internal map.

$\texttt{getProto}^\sharp \in Object^\sharp \rightarrow BValue^\sharp$

$\texttt{getProto}^\sharp(\hat{o}) = \pi_2(\hat{o})(\texttt{"proto"})$

3.4.7 $\texttt{getClass}^\sharp$

$\texttt{getClass}^\sharp$ retrieves the `"class"` property from an object's internal map.

$\texttt{getClass}^\sharp \in Object^\sharp \rightarrow Class$

$\texttt{getClass}^\sharp(\hat{o}) = \pi_2(\hat{o})(\texttt{"class"})$

3.4.8 $\texttt{id}(\cdot)$

$\texttt{id}(\cdot)$ takes an expression or statement and returns a unique identifier. It is used by the abstract interpreter to create $Address^\sharp$ and abstract traces.

$\texttt{id}(\cdot) \in (Exp + Stmt) \rightarrow Identifier$

3.4.9 $\texttt{initState}^\sharp$

$\texttt{initState}^\sharp$ takes a program and returns the initial abstract state, containing the initial environment and store.

$\texttt{initState}^\sharp \in Decl \rightarrow State^\sharp$

$\texttt{initState}^\sharp(d) = (d,\ \hat{\rho},\ \hat{\sigma},\ \widehat{\textbf{haltK}})$

where $\hat{\rho}$ and $\hat{\sigma}$ are abstracted versions of the concrete versions described in `builtin.pdf`.

3.4.10 $\texttt{inject}^\sharp$

$\texttt{inject}^\sharp$ takes a single component of an abstract base value and returns an abstract base value where everything is bottom except the specified component.

$$\texttt{inject}^\sharp \in (\mathit{Num}^\sharp + \mathcal{P}(\mathit{Bool}) + \mathit{String}^\sharp + \mathcal{P}(\mathit{Address}^\sharp) + \mathcal{P}(\mathit{Closure}^\sharp) + \mathcal{P}(\{\textsf{null}\}) + \mathcal{P}(\{\textsf{undef}\})) \to \mathit{BValue}^\sharp$$

$$\texttt{inject}^\sharp\,(\hat{n}) = (\hat{n}, \emptyset, \bot_{\widehat{str}}, \emptyset, \emptyset, \emptyset, \emptyset)$$

$$\texttt{inject}^\sharp\left(\hat{b}\right) = (\bot_{\hat{n}}, \hat{b}, \bot_{\widehat{str}}, \emptyset, \emptyset, \emptyset, \emptyset)$$

$$\texttt{inject}^\sharp\left(\widehat{str}\right) = (\bot_{\hat{n}}, \emptyset, \widehat{str}, \emptyset, \emptyset, \emptyset, \emptyset)$$

$$\texttt{inject}^\sharp\,(\bar{\hat{a}}) = (\bot_{\hat{n}}, \emptyset, \bot_{\widehat{str}}, \bar{\hat{a}}, \emptyset, \emptyset, \emptyset)$$

$$\texttt{inject}^\sharp\,(\textsf{null}) = (\bot_{\hat{n}}, \emptyset, \bot_{\widehat{str}}, \emptyset, \emptyset, \{\textsf{null}\}, \emptyset)$$

$$\texttt{inject}^\sharp\,(\textsf{undef}) = (\bot_{\hat{n}}, \emptyset, \bot_{\widehat{str}}, \emptyset, \emptyset, \emptyset, \{\textsf{undef}\})$$

3.4.11 $\texttt{lookup}^\sharp$

$\texttt{lookup}^\sharp$ looks up the value of a property in an object, going up the prototype chain if necessary. If the property is potentially absent, $\texttt{lookup}^\sharp$ joins the value of the property with the value returned by looking up the prototype chain.

$$\texttt{lookup}^\sharp \in \mathcal{P}(\mathit{Address}^\sharp) \times \mathit{String}^\sharp \times \mathit{Store}^\sharp \to \mathit{BValue}^\sharp$$

$$\texttt{lookup}^\sharp(\bar{\hat{a}}, \widehat{str}, \hat{\sigma}) = \bigsqcup \left\{\ \widehat{bv} \in \texttt{look}^\sharp(\hat{o}, \widehat{str}, \hat{\sigma}) \ \mid\ \hat{a} \in \bar{\hat{a}},\ \hat{o} = \hat{\sigma}(\hat{a})\ \right\}$$

$\texttt{look}^\sharp$ is used by $\texttt{lookup}^\sharp$ to look up a property in an individual object; it returns the set of possible values (including those from looking up the prototype chain when necessary). $\texttt{look}^\sharp$ searches the given object for all properties comparable to the given abstract string and joins their values.

$$\texttt{look}^\sharp \in \mathit{Object}^\sharp \times \mathit{String}^\sharp \times \mathit{Store}^\sharp \to \mathcal{P}(\mathit{BValue}^\sharp)$$

$$\texttt{look}^\sharp(\hat{o}, \widehat{str}, \hat{\sigma}) = \mathit{local} \cup \mathit{chain} \cup \mathit{fin} \text{ where}$$

$$\mathit{local} = \bigsqcup \left\{\ \hat{o}(\widehat{str}') \ \mid\ \widehat{str}' \in \texttt{dom}(\hat{o}),\ \widehat{str} \sqsubseteq \widehat{str}' \vee \widehat{str}' \sqsubseteq \widehat{str}\ \right\}$$

$$\mathit{chain} = \begin{cases} \bigcup \left\{\ \texttt{look}^\sharp(\hat{o}', \widehat{str}, \hat{\sigma}) \ \mid\ \hat{a} \in \pi_{\hat{a}}(\texttt{getProto}^\sharp(\hat{o})),\ \hat{o}' = \hat{\sigma}(\hat{a})\ \right\} & \text{if } \widehat{str} \notin \pi_3(\hat{o}) \\ \emptyset & \text{otherwise} \end{cases}$$

$$\mathit{fin} = \begin{cases} \texttt{inject}^\sharp(\textsf{undef}) & \text{if } \widehat{str} \notin \pi_3(\hat{o}),\ \pi_{\widehat{null}}(\texttt{getProto}^\sharp(\hat{o})) \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases}$$

3.4.12 $\texttt{nextK}^\sharp$

$\texttt{nextK}^\sharp$ takes abstract continuations containing other abstract continuations (i.e., any continuation other than $\widehat{\textsf{haltK}}$ and $\widehat{\textsf{addrK}}$) and returns the enclosed continuation. It is, in essence, "popping" the continuation stack.

$$\texttt{nextK}^\sharp \in \mathit{Kont}^\sharp \setminus \{\widehat{\textsf{haltK}}, \widehat{\textsf{addrK}}\} \to \mathit{Kont}^\sharp$$

$$\texttt{nextK}^\sharp(\hat{\kappa}) = \hat{\kappa}_c \quad \text{where}$$

$$\hat{\kappa} = \langle \mathit{name} \rangle \ldots \hat{\kappa}_c$$

3.4.13 $\texttt{objKeys}^\sharp$

$\texttt{objKeys}^\sharp$ returns the set of properties that may be present in the referenced objects (we implicitly ignore properties with the **noenum** attribute when computing the object map's domain using $\texttt{dom}$).

$$\texttt{objKeys}^\sharp \in \mathit{BValue}^\sharp \times \mathit{Store}^\sharp \to \overrightarrow{\mathit{String}^\sharp}_i$$

$$\texttt{objKeys}^\sharp(\widehat{bv}, \hat{\sigma}) = \left\{\ \widehat{str} \in \texttt{dom}(\hat{\sigma}(\hat{a})) \ \mid\ \hat{a} \in \pi_{\hat{a}}(\widehat{bv})\ \right\}$$

3.4.14 $\texttt{setConstr}^\sharp$

`setConstr` is used to set the internal `"constructor"` property of the argument (which is guaranteed by the translator to be a singleton address of an object) to **true**.

$$\texttt{setConstr}^\sharp \in \mathit{Store}^\sharp \times \mathit{BValue}^\sharp \to \mathit{Store}^\sharp$$

$$\texttt{setConstr}^\sharp(\hat{\sigma}, \widehat{bv}) = \hat{\sigma}' \quad \text{where}$$

$$\{\hat{a}\} = \pi_{\hat{a}}(\widehat{bv})$$

$$\mathit{addc}(\hat{o}) = (\pi_1(\hat{o}), \pi_2(\hat{o})[\texttt{"constructor"} \mapsto \textsf{\textbf{true}}], \pi_3(\hat{o}))$$

$$\hat{\sigma}' = \hat{\sigma}[\hat{a} \mapsto \mathit{addc}(\hat{\sigma}(\hat{a}))]$$

3.4.15 $\texttt{specialK}^\sharp$

$\texttt{specialK}^\sharp$ is used by the default exceptional/jump value propagation rule that pops the continuation stack until it finds a continuation such that there is a different rule used to handle that continuation. Given a value, $\texttt{specialK}^\sharp$ returns the set of continuations that should not be popped when propagating that type of value because there are other rules that handle them.

$$\texttt{specialK}^\sharp \in \mathit{Value}^\sharp \to \mathcal{P}(\mathit{Kont}^\sharp)$$

$$\texttt{specialK}^\sharp(\hat{v}) =$$

$$\begin{cases} \mathit{Kont}^\sharp & \text{if } \hat{v} \in \mathit{BValue}^\sharp \\ \left\{\widehat{\textsf{\textbf{haltK}}}, \widehat{\textsf{\textbf{addrK}}}, \widehat{\textsf{\textbf{tryK}}}, \widehat{\textsf{\textbf{catchK}}}, \widehat{\textsf{\textbf{retK}}}\right\} & \text{if } \hat{v} \in \mathit{EValue}^\sharp \\ \left\{\widehat{\textsf{\textbf{haltK}}}, \widehat{\textsf{\textbf{addrK}}}, \widehat{\textsf{\textbf{tryK}}}, \widehat{\textsf{\textbf{catchK}}}, \widehat{\textsf{\textbf{lblK}}}\right\} & \text{if } \hat{v} \in \mathit{JValue}^\sharp \end{cases}$$

3.4.16 $\texttt{toObj}^\sharp$

$\texttt{toObj}^\sharp$ attempts to convert a value into an object, allocating a new object if necessary.

$\texttt{toObj}^\sharp \in \mathit{BValue}^\sharp \times \mathit{Variable} \times \mathit{Env}^\sharp \times \mathit{Store}^\sharp \rightarrow \mathcal{O}(\mathit{BValue}^\sharp \times \mathit{Store}^\sharp) \times \mathcal{O}(\mathit{EValue}^\sharp)$

$\texttt{toObj}^\sharp(\widehat{bv}, x, \hat{\rho}, \hat{\sigma}) = (\mathit{noexc}, \mathit{exc}) \quad \text{where}$

$$\mathit{noexc} = \begin{cases} \bigsqcup \overline{(\widehat{bv}, \hat{\sigma})} & \text{if } |\overline{(\widehat{bv}, \hat{\sigma})}| > 0 \\ \emptyset & \text{otherwise} \end{cases}$$

$$\mathit{exc} = \begin{cases} \textbf{exc } \alpha_{str}(\texttt{"TypeError"}) & \text{if } \pi_{\widehat{null}}(\widehat{bv}) \bigcup \pi_{\widehat{undef}}(\widehat{bv}) \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases}$$

$$\overline{(\widehat{bv}, \hat{\sigma})} = \mathit{addr} \cup \mathit{num} \cup \mathit{bool} \cup \mathit{string}$$

$$(\hat{\sigma}_n, \widehat{bv}_n) = \texttt{allocObj}^\sharp(\texttt{inject}^\sharp(\widehat{\texttt{Number\_Addr}}), \hat{\sigma})$$

$$(\hat{\sigma}_b, \widehat{bv}_b) = \texttt{allocObj}^\sharp(\texttt{inject}^\sharp(\widehat{\texttt{Boolean\_Addr}}), \hat{\sigma})$$

$$(\hat{\sigma}_{str}, \widehat{bv}_{str}) = \texttt{allocObj}^\sharp(\texttt{inject}^\sharp(\widehat{\texttt{String\_Addr}}), \hat{\sigma})$$

$$\mathit{extras} = \begin{cases} [\texttt{"length"} \mapsto \top_{\hat{n}}, \top_{\hat{n}} \mapsto \top_{\widehat{str}}] & \text{if } \pi_{\widehat{str}}(\widehat{bv}) \neq \bot_{\widehat{str}} \\ \emptyset & \text{otherwise} \end{cases}$$

$$\mathit{addvalue}(\hat{o}) = (\pi_1(\hat{o}), \pi_2(\hat{o})[\texttt{"value"} \mapsto \widehat{bv} \sqcup \pi_2(\hat{o})(\texttt{"value"})][\mathit{extras}], \pi_3(\hat{o}))$$

$$\mathit{addr} = \begin{cases} (\texttt{inject}^\sharp(\pi_{\hat{a}}(\widehat{bv})), \hat{\sigma}) & \text{if } |\pi_{\hat{a}}(\widehat{bv})| > 0 \\ \emptyset & \text{otherwise} \end{cases}$$

$$\mathit{num} = \begin{cases} (\widehat{bv}_n, \texttt{update}^\sharp(\hat{\sigma}', \hat{\rho}(x), \widehat{bv}_n)) & \text{if } \pi_{\hat{n}}(\widehat{bv}) \neq \bot_{\hat{n}},\ \hat{a} = \pi_{\hat{a}}(\widehat{bv}_n),\ \hat{o} = \hat{\sigma}_n(\hat{a}),\ \hat{\sigma}' = \hat{\sigma}_n[\hat{a} \mapsto \mathit{addvalue}(\hat{o})] \\ \emptyset & \text{otherwise} \end{cases}$$

$$\mathit{bool} = \begin{cases} (\widehat{bv}_b, \texttt{update}^\sharp(\hat{\sigma}', \hat{\rho}(x), \widehat{bv}_b)) & \text{if } \pi_{\hat{b}}(\widehat{bv}) \neq \emptyset,\ \hat{a} = \pi_{\hat{a}}(\widehat{bv}_b),\ \hat{o} = \hat{\sigma}_b(\hat{a}),\ \hat{\sigma}' = \hat{\sigma}_b[\hat{a} \mapsto \mathit{addvalue}(\hat{o})] \\ \emptyset & \text{otherwise} \end{cases}$$

$$\mathit{string} = \begin{cases} (\widehat{bv}_{str}, \texttt{update}^\sharp(\hat{\sigma}', \hat{\rho}(x), \widehat{bv}_{str})) & \text{if } \pi_{\widehat{str}}(\widehat{bv}) \neq \bot_{\widehat{str}},\ \hat{a} = \pi_{\hat{a}}(\widehat{bv}_{str}),\ \hat{o} = \hat{\sigma}_{str}(\hat{a}),\ \hat{\sigma}' = \hat{\sigma}_{str}[\hat{a} \mapsto \mathit{addvalue}(\hat{o})] \\ \emptyset & \text{otherwise} \end{cases}$$

3.4.17 $\texttt{update}^\sharp$

$\texttt{update}^\sharp$ is used to update an abstract store. The update will be strong (replacing the old value) or weak (joining with the old value) depending on whether the abstract address represents a single concrete address or potentially many concrete addresses.

$\texttt{update}^\sharp \in \mathit{Store}^\sharp \times \mathcal{P}(\mathit{Address}^\sharp) \times \mathit{BValue}^\sharp \rightarrow \mathit{Store}^\sharp$

$\texttt{update}^\sharp(\hat{\sigma}, \bar{\hat{a}}, \widehat{bv}) =$

$$\begin{cases} \hat{\sigma}[\hat{a} \mapsto \widehat{bv}] & \text{if } \bar{\hat{a}} = \{\hat{a}\},\ |\gamma_a(\hat{a})| = 1 \\ \hat{\sigma}[\hat{a} \mapsto \widehat{bv} \sqcup \hat{\sigma}(\hat{a})] & \text{otherwise, for } \hat{a} \in \bar{\hat{a}} \end{cases}$$

3.4.18 $\texttt{updateObj}^\sharp$

$\texttt{updateObj}^\sharp$ updates an object's property (adding the property if it doesn't already exist). $\texttt{updateObj}^\sharp$ returns a tuple; the first component is the new store when $\texttt{updateObj}^\sharp$ does not raise an exception, and the second component is for the case when it does. $\texttt{updateObj}^\sharp$ does not update any property whose attributes include **noupdate**. Array objects also have special rules involving property updates that we enforce here.

Note: Precisely tracking the Array rules is complex, and we can get most of the payoff with a sound but simpler and less precise approach, namely: we set the abstract value of the `"length"` property to the closest approximation in the abstract number domain of a generic unsigned 32-bit integer value, and guarantee that it will never change. Thus we only need to check for two special cases here: (1) range errors, and (2) potentially deleting existing properties when `"length"` is updated to a value less than its current value.

$$\texttt{updateObj}^\sharp \in \mathit{BValue}^\sharp \times \mathit{BValue}^\sharp \times \mathit{BValue}^\sharp \times \mathit{Store}^\sharp \to \mathcal{O}(\mathit{BValue}^\sharp \times \mathit{Store}^\sharp) \times \mathcal{O}(\mathit{EValue}^\sharp)$$

$$\texttt{updateObj}^\sharp(\widehat{bv}_1, \widehat{bv}_2, \widehat{bv}_3, \hat{\sigma}) = (\mathit{noexc}, \mathit{exc}) \quad \text{where}$$

$$\widehat{str} = \pi_{\widehat{str}}(\widehat{bv}_2)$$

$$\mathit{maybeArray?} = \exists \hat{a} \in \pi_{\hat{a}}(\widehat{bv}_1)\,.\,\texttt{getClass}^\sharp(\hat{a}) = \textsf{\textbf{array}}$$

$$\mathit{notU32?}(\widehat{bv}) = \widehat{bv} \text{ may not represent an unsigned 32-bit integer}$$

$$\mathit{newobjs} = \left\{ (\hat{a}, \hat{o}) \;\middle|\; \hat{a} \in \pi_{\hat{a}}(\widehat{bv}_1),\ \hat{o} = \texttt{insert}^\sharp(\hat{\sigma}(\hat{a}), \widehat{str}, \widehat{bv}_3) \right\}$$

$$\mathit{noexc} = \begin{cases} (bv_3, \hat{\sigma}[\hat{a} \mapsto \hat{o}]) & \text{if } \mathit{newobjs} = \{(\hat{a}, \hat{o})\},\ |\gamma_a(\hat{a})| = 1,\ |\gamma_{str}(\widehat{str})| = 1 \\ (bv_3, \hat{\sigma}[\overrightarrow{\hat{a}_i \mapsto \hat{\sigma}(\hat{a}_i) \sqcup \hat{o}_i}]) & \text{otherwise if } |\mathit{newobj}| > 0,\ (\hat{a}_i, \hat{o}_i) \in \mathit{newobjs} \\ \emptyset & \text{otherwise} \end{cases}$$

$$\mathit{exc} = \begin{cases} \textsf{\textbf{exc}}\ \alpha_{str}(\texttt{"TypeError"}) & \text{if } \pi_{\widehat{null}}(\widehat{bv}_1) \bigcup \pi_{\widehat{undef}}(\widehat{bv}_1) \neq \emptyset \\ \textsf{\textbf{exc}}\ \alpha_{str}(\texttt{"RangeError"}) & \text{if } \pi_{\widehat{null}}(\widehat{bv}_1) \bigcup \pi_{\widehat{undef}}(\widehat{bv}_1) = \emptyset,\ \mathit{maybeArray?},\ \texttt{"length"} \in \gamma_{str}(\widehat{str}),\ \mathit{notU32}(\widehat{bv}_3) \\ \emptyset & \text{otherwise} \end{cases}$$

$\texttt{insert}^\sharp$ is used to insert a property into an individual object, updating the set of definitely-present properties as appropriate. Note that for weak updates the result of $\texttt{insert}^\sharp$ is joined with the old object.

$$\texttt{insert}^\sharp \in \mathit{Object}^\sharp \times \mathit{String}^\sharp \times \mathit{BValue}^\sharp \to \mathit{Object}^\sharp$$

$$\texttt{insert}^\sharp(\hat{o}, \widehat{str}, \widehat{bv}) =$$

$$\begin{cases} \hat{o} \sqcup \hat{o}' & \text{if } \mathit{array?},\ \gamma_{str}(\widehat{str}) = \{\texttt{"length"}\} \\ \left(\pi_1(\hat{o})[\widehat{str} \mapsto \widehat{bv}], \pi_2(\hat{o}), \pi_3(\hat{o}) \cup \widehat{str}\right) & \text{if } \gamma_{str}(\widehat{str}) = \{str\},\ (\neg \mathit{array?} \vee str \neq \texttt{"length"}) \\ \hat{o}'[\widehat{str} \mapsto \widehat{bv}] & \text{if } \mathit{array?},\ |\gamma_{str}(\widehat{str})| > 1,\ \texttt{"length"} \in \gamma_{str}(\widehat{str}),\ \mathit{maybeU32}(\widehat{bv}) \\ \hat{o}[\widehat{str} \mapsto \widehat{bv}] & \text{otherwise} \end{cases}$$

where

$$\mathit{array?} = \pi_2(\hat{o})(\texttt{"class"}) = \textsf{\textbf{array}}$$

$$\mathit{maybeU32}(\widehat{bv}) = \widehat{bv} \text{ may represent an unsigned 32-bit integer}$$

$$\hat{o}' = \hat{o} - \left\{ \widehat{str} \;\middle|\; \widehat{str} \in \texttt{dom}(\hat{o}),\ \mathit{maybeU32}(\textsf{\textbf{tonum}}\ \widehat{str}) \right\}$$