Lwnn Abstract Semantics

CS 260

1 Lwnn Abstract Syntax

$$n \in \mathbb{Z} \qquad b \in \mathit{Bool} \qquad \mathit{str} \in \mathit{String} \qquad x \in \mathit{Variable}$$
$$\mathit{cn} \in \mathit{ClassName} \qquad \mathit{mn} \in \mathit{MethodName}$$

$$\begin{aligned}
p \in \mathit{Program} &::= \overrightarrow{\mathit{class}} \\
\mathit{class} \in \mathit{Class} &::= \textbf{class}\ \mathit{cn}_1\ \textbf{extends}\ \mathit{cn}_2\ \left\{\ \textbf{fields}\ \overrightarrow{x : \tau}\ \cdot\ \textbf{methods}\ \overrightarrow{m}\ \right\} \\
\tau \in \mathit{Type} &::= \texttt{int} \mid \texttt{bool} \mid \texttt{string} \mid \texttt{null} \mid \mathit{cn} \\
m \in \mathit{Method} &::= \textbf{def}\ \mathit{mn}(\overrightarrow{x : \tau}) : \tau_{ret}\ \{\ \vec{s} \cdot \textbf{return}\ e\ \} \\
s \in \mathit{Stmt} &::= x := e \mid e_1.x := e_2 \mid x := e.\mathit{mn}(\vec{e}) \mid x := \textbf{new}\ \mathit{cn}(\vec{e}) \\
&\quad \mid \textbf{if}\ e\ \overrightarrow{s_1}\ \textbf{else}\ \overrightarrow{s_2} \mid \textbf{while}\ e\ \vec{s} \\
e \in \mathit{Exp} &::= \overline{n} \mid \overline{b} \mid \overline{\mathit{str}} \mid \textbf{null} \mid x \mid e.x \mid e_1 \oplus e_2 \\
\oplus \in \mathit{BinaryOp} &::= + \mid - \mid \times \mid \div \mid < \mid \leq \mid \wedge \mid \vee \mid = \mid \neq
\end{aligned}$$

This serves as a reminder—the abstract syntax is exactly the same as it was for the concrete semantics.

2 Lwnn Abstract Semantics

We describe the semantic domains that constitute an abstract state (Section 2.1), abstract state transition rules (Section 2.2), and the helper functions used by the abstract transition rules (Section 2.3).

2.1 Abstract Semantic Domains

$$\hat{n} \in \mathbb{Z}^\sharp \qquad \hat{b} \in Bool^\sharp \qquad \widehat{str} \in String^\sharp \qquad \hat{a} \in Address^\sharp \qquad \hat{\oplus} \in BinaryOp^\sharp$$

$$\begin{aligned}
\hat{\varsigma} \in State^\sharp &= ClassDefs \times Stmt^? \times Locals^\sharp \times Heap^\sharp \times Kont^{\sharp\star} \\
\theta \in ClassDefs &= ClassName \to ((Variable \to Type) \times (MethodName \to Method)) \\
\hat{\rho} \in Locals^\sharp &= Variable \to Value^\sharp \\
\hat{\sigma} \in Heap^\sharp &= Address^\sharp \to \left(Object^\sharp \uplus \mathcal{P}\left(Kont^{\sharp\star}\right)\right) \\
\hat{r} \in Reference^\sharp &= \mathcal{P}\left(Address^\sharp \cup \{\textbf{null}\}\right) \\
\hat{v} \in Value^\sharp &= \mathbb{Z}^\sharp \uplus Bool^\sharp \uplus String^\sharp \uplus Reference^\sharp \\
\hat{o} \in Object^\sharp &= ClassName \times \left(Variable \to Value^\sharp\right) \\
\hat{\kappa} \in Kont^\sharp &= \widehat{\textbf{stmtK}}\ Stmt \uplus \widehat{\textbf{whileK}}\ Exp \times Stmt^\star \uplus \widehat{\textbf{retK}}\ Variable \times Exp \times Locals^\sharp \uplus \widehat{\textbf{finK}}\ Address^\sharp
\end{aligned}$$

Domains Summary. We leave the abstract number, boolean, string, and address domains unspecified. The abstract boolean operators depend on the specific abstractions chosen for these domains. We augment the $Heap^\sharp$ domain to map to sets of continuation stacks and add a semantic continuation $\widehat{\textbf{finK}}$; these are used to provide a level of indirection for handling method calls that is necessary for computability. An abstract reference is a set containing abstract addresses and/or **null**—we do this because we are over-approximating the concrete semantics, in which an object reference would be only a single address or **null**.

2.2 Abstract Transition Rules

Table 1: The abstract transition relation. Each rule describes how to take one abstract state $(\theta, s^?, \hat{\rho}, \hat{\sigma}, \vec{\hat{\kappa}})$ to the next abstract state $(\theta, s^?_{new}, \hat{\rho}_{new}, \hat{\sigma}_{new}, \overrightarrow{\hat{\kappa}_{new}})$, where $\vec{\hat{\kappa}} = \hat{\kappa} \cdot \overrightarrow{\hat{\kappa}_1}$. The $s^?$ notation means a statement may or may not exist; we use $\bullet$ to indicate that there is no statement. The $\cdot$ operator used for the continuation stacks indicates appending sequences; thus $\hat{\kappa}$ is the top of the continuation stack in the source state and $\overrightarrow{\hat{\kappa}_1}$ is the rest of that continuation stack.

no.	$s^?$	**premises**	$s^?_{new}$	$\hat{\rho}_{new}$	$\hat{\sigma}_{new}$	$\overrightarrow{\hat{\kappa}_{new}}$
1	$x := e$	$[\![e]\!]^\sharp = \hat{v}$	$\bullet$	$\hat{\rho}[x \mapsto \hat{v}]$	$\hat{\sigma}$	$\vec{\hat{\kappa}}$
2	$e_1.x := e_2$	$\hat{\sigma}_1 = \texttt{update}^\sharp(\hat{\sigma}, [\![e_1]\!]^\sharp, x, [\![e_2]\!]^\sharp)$	$\bullet$	$\hat{\rho}$	$\hat{\sigma}_1$	$\vec{\hat{\kappa}}$
3	$x := e.mn(\vec{e})$	$\left(\hat{\rho}_1, \hat{\sigma}_1, \overrightarrow{\hat{\kappa}_2}\right) \in \texttt{call}^\sharp\left(\theta, x, [\![e]\!]^\sharp, \hat{\sigma}, mn, \overrightarrow{[\![e]\!]^\sharp}, \hat{\rho}, \vec{\hat{\kappa}}\right)$	$\bullet$	$\hat{\rho}_1$	$\hat{\sigma}_1$	$\overrightarrow{\hat{\kappa}_2}$
4	$x := \textbf{new}\ cn(\vec{e})$	$\left(\hat{\rho}_1, \hat{\sigma}_1, \overrightarrow{\hat{\kappa}_2}\right) = \texttt{construct}^\sharp\left(\theta, x, cn, \overrightarrow{[\![e]\!]^\sharp}, \hat{\rho}, \hat{\sigma}, \vec{\hat{\kappa}}\right)$	$\bullet$	$\hat{\rho}_1$	$\hat{\sigma}_1$	$\overrightarrow{\hat{\kappa}_2}$
5	$\textbf{if}\ e\ \overrightarrow{s_1}\ \textbf{else}\ \overrightarrow{s_2}$	$\textbf{true} \in \gamma_b([\![e]\!]^\sharp)$	$\bullet$	$\hat{\rho}$	$\hat{\sigma}$	$\texttt{toSK}^\sharp(\overrightarrow{s_1}) \cdot \vec{\hat{\kappa}}$
6	$\textbf{if}\ e\ \overrightarrow{s_1}\ \textbf{else}\ \overrightarrow{s_2}$	$\textbf{false} \in \gamma_b([\![e]\!]^\sharp)$	$\bullet$	$\hat{\rho}$	$\hat{\sigma}$	$\texttt{toSK}^\sharp(\overrightarrow{s_2}) \cdot \vec{\hat{\kappa}}$
7	$\textbf{while}\ e\ \vec{s}$	$\textbf{true} \in \gamma_b([\![e]\!]^\sharp)$	$\bullet$	$\hat{\rho}$	$\hat{\sigma}$	$\texttt{toSK}^\sharp(\vec{s}) \cdot \widehat{\textbf{whileK}}(e, \vec{s}) \cdot \vec{\hat{\kappa}}$
8	$\textbf{while}\ e\ \vec{s}$	$\textbf{false} \in \gamma_b([\![e]\!]^\sharp)$	$\bullet$	$\hat{\rho}$	$\hat{\sigma}$	$\vec{\hat{\kappa}}$
9	$\bullet$	$\hat{\kappa} = \widehat{\textbf{finK}}(\hat{a}), \widehat{\textbf{retK}}(x, e, \hat{\rho}_1) \cdot \overrightarrow{\hat{\kappa}_2} \in \hat{\sigma}(\hat{a}), [\![e]\!]^\sharp = \hat{v}$	$\bullet$	$\hat{\rho}_1[x \mapsto \hat{v}]$	$\hat{\sigma}$	$\overrightarrow{\hat{\kappa}_2}$
10	$\bullet$	$\hat{\kappa} = \widehat{\textbf{stmtK}}(s_1)$	s_1	$\hat{\rho}$	$\hat{\sigma}$	$\overrightarrow{\hat{\kappa}_1}$
11	$\bullet$	$\hat{\kappa} = \widehat{\textbf{whileK}}(e, \vec{s}), \textbf{true} \in \gamma_b([\![e]\!]^\sharp)$	$\bullet$	$\hat{\rho}$	$\hat{\sigma}$	$\texttt{toSK}^\sharp(\vec{s}) \cdot \vec{\hat{\kappa}}$
12	$\bullet$	$\hat{\kappa} = \widehat{\textbf{whileK}}(e, \vec{s}), \textbf{false} \in \gamma_b([\![e]\!]^\sharp)$	$\bullet$	$\hat{\rho}$	$\hat{\sigma}$	$\overrightarrow{\hat{\kappa}_1}$

Transitions Summary. Rules 5–8 and 11–12 use γ_b, the boolean concretization operator that maps an abstract boolean value to the corresponding set of concrete boolean values. In rules 2–3 the helper functions store the current continuation stack inside $\hat{\sigma}_1$ for a method/constructor call, and rule 9 restores the continuation stack when returning from the callee.

2.3 Abstract Helper Functions

We describe the helper functions used by the abstract transition rules. The functions are listed in alphabetical order. Note that in several places we implicitly ignore the possibility that an abstract reference value may contain **null**; since these operations on **null** are undefined in the concrete semantics, it is sound to ignore them in the abstract semantics.

2.3.1 $\eta^\sharp(e, \hat{\rho}, \hat{\sigma})$ **a.k.a.** $[\![e]\!]^\sharp$

This function describes how to evaluate expressions to abstract values. It uses the number abstaction function α_n, the boolean abstraction function α_b, the string abstraction function α_{str}, and the reference abstraction function α_r. Field access uses the helper function $\texttt{lookup}^\sharp$. The abstract binary operators are left unspecified because they depend on the abstractions chosen for the abstract value domains.

$\eta^\sharp : Exp \times Locals^\sharp \times Heap^\sharp \to Value^\sharp$

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

$$\begin{cases} \alpha_n(\overline{n}) & \text{if } e = \overline{n} \\ \alpha_b(\overline{b}) & \text{if } e = \overline{b} \\ \alpha_{str}(\overline{str}) & \text{if } e = \overline{str} \\ \alpha_r(\textbf{null}) & \text{if } e = \textbf{null} \\ \hat{\rho}(x) & \text{if } e = x \\ \texttt{lookup}^\sharp([\![e_1]\!]^\sharp, x, \hat{\sigma}) & \text{if } e = e_1.x \\ [\![e_1]\!]^\sharp \mathbin{\hat{\oplus}} [\![e_2]\!]^\sharp & \text{if } e = e_1 \oplus e_2 \end{cases}$$

2.3.2 $\texttt{call}^\sharp$

This function describes how to abstractly process a method call. It returns a set of $(Locals^\sharp, Heap^\sharp, Kont^{\sharp\star})$ pairs because (due to inheritance and subtype polymorphism) there could be more than one possible method being called. The returned $\hat{\sigma}_1$'s also contain the current continuation stack, which will be restored once the callee returns; this extra level of indirection is necessary for computability. Note that the address of the continuation stack (i.e., $\hat{a}_\kappa$) is always weakly updated in the stores.

$\texttt{call}^\sharp \in ClassDefs \times Variable \times \mathcal{P}(Address^\sharp) \times Heap^\sharp \times MethodName \times Value^{\sharp\star} \times Locals^\sharp \times Kont^{\sharp\star} \to \mathcal{P}(Locals^\sharp \times Heap^\sharp \times Kont^{\sharp\star})$

$\texttt{call}^\sharp\left(\theta, x, \overline{\hat{a}}, \hat{\sigma}, mn, \vec{\hat{v}}, \hat{\rho}, \vec{\hat{\kappa}}\right) = \overline{\left(\hat{\rho}_1, \hat{\sigma}_1, \overrightarrow{\hat{\kappa}_1}\right)}$ where

$\overline{(\hat{a}, m)} = \left\{ (\hat{a},\ methods(mn)) \mid \hat{a} \in \overline{\hat{a}},\ cn = \pi_1(\sigma(\hat{a})),\ methods = \pi_2(\theta(cn)) \right\}$

$\overline{\left(\hat{\rho}_1, \hat{\sigma}_1, \overrightarrow{\hat{\kappa}_1}\right)} = \left\{ \left(\hat{\rho}_{\hat{a}}, \hat{\sigma}_{\hat{a}}, \overrightarrow{\hat{\kappa}_{\hat{a}}}\right) \;\middle|\; \left(\hat{a},\ \textbf{def}\ mn(\overrightarrow{x : \tau}) : \tau_{ret}\ \{\ \vec{s} \cdot \textbf{return}\ e\ \}\right) \in \overline{(\hat{a}, m)} \right\}$ where

$\hat{a}_\kappa$ depends on the heap model

$\hat{\rho}_{\hat{a}} = [\texttt{self} \mapsto \hat{a}] \cup [\ x_i \mapsto \hat{v} \mid 0 \le i < |\vec{\hat{v}}| \implies \hat{v} = \hat{v}_i,\ |\vec{\hat{v}}| \le i < |\overrightarrow{x : \tau}| \implies \hat{v} = \texttt{defaultvalue}^\sharp(\tau_i)\ ]$

$\hat{\sigma}_{\hat{a}} = \hat{\sigma} \cup \left[\hat{a}_\kappa \mapsto \left\{ \widehat{\textbf{retK}}(x,\ e,\ \hat{\rho}) \cdot \vec{\hat{\kappa}} \right\}\right]$

$\overrightarrow{\hat{\kappa}_{\hat{a}}} = \texttt{toSK}^\sharp(\vec{s}) \cdot \widehat{\textbf{finK}}(\hat{a}_\kappa)$

2.3.3 $\texttt{construct}^\sharp$

This function describes how to create a new abstract object. It leaves determination of the abstract address at which to allocate the object unspecified; this will depend on the heap model used by the analysis. That abstract address may be *strong* (i.e., correspond to a single concrete address) or *weak* (otherwise); in the first case $\hat{\sigma}_1$ is updated with the new object, in the second case $\hat{\sigma}_1$ is updated

with the join of the new object and any object currently allocated at that address—since the set of abstract addresses must be finite, the analysis may have to reuse the same abstract address for different objects. The γ_a operator is the abstract address concretization operator; note that the *implementation* of this helper function *should not* actually use the concretization operator (which would be uncomputable). As with $\texttt{call}^\sharp$, the address of the continuation stack (i.e., $\hat{a}_\kappa$) is always weakly updated in the stores.

$\texttt{construct}^\sharp \in \mathit{ClassDefs} \times \mathit{Variable} \times \mathit{ClassName} \times \mathit{Value}^{\sharp\star} \times \mathit{Locals}^\sharp \times \mathit{Heap}^\sharp \times \mathit{Kont}^{\sharp\star} \to \mathit{Locals}^\sharp \times \mathit{Heap}^\sharp \times \mathit{Kont}^{\sharp\star}$

$\texttt{construct}^\sharp\left(\theta, x, cn, \vec{\hat{v}}, \hat{\rho}, \hat{\sigma}, \vec{\hat{k}}\right) = \left(\hat{\rho}_1, \hat{\sigma}_2, \overrightarrow{\hat{k}_1}\right)$ where

$\hat{a}, \hat{a}_\kappa$ depend on the heap model

$\mathit{flds} = [\ x \mapsto \hat{v}\ \mid\ \pi_1(\theta(cn))(x) = \tau,\ \texttt{defaultvalue}^\sharp(\tau) = v\ ]$

$\hat{o} = (cn, \mathit{flds})$

$\mathit{methods} = \pi_2(\theta(cn))$

$\mathit{methods}(cn) = \textbf{def}\ cn(\overrightarrow{x : \tau}) : \tau_{ret}\ \{\ \vec{s} \cdot \textbf{return}\ \texttt{self}\ \}$

$\hat{\rho}_1 = [\texttt{self} \mapsto \hat{a}] \cup [\ x_i \mapsto \hat{v}\ \mid\ 0 \le i < |\vec{\hat{v}}| \implies \hat{v} = \hat{v}_i,\ |\vec{\hat{v}}| \le i < |\overrightarrow{x : \tau}| \implies \hat{v} = \texttt{defaultvalue}^\sharp(\tau_i)\ ]$

$$\hat{\sigma}_1 = \begin{cases} \hat{\sigma}[\hat{a} \mapsto \hat{o}] & \text{if } |\gamma_a(\hat{a})| = 1 \\ \hat{\sigma}[\hat{a} \mapsto \hat{o} \sqcup \hat{\sigma}(\hat{a})] & \text{otherwise} \end{cases}$$

$\hat{\sigma}_2 = \hat{\sigma}_1 \cup \left[\hat{a}_\kappa \mapsto \left\{\widehat{\textbf{retK}}(x, \texttt{self}, \hat{\rho}) \cdot \vec{\hat{k}}\right\}\right]$

$\overrightarrow{\hat{k}_1} = \texttt{toSK}^\sharp(\vec{s}) \cdot \widehat{\textbf{finK}}(\hat{a}_\kappa)$

2.3.4 $\texttt{defaultvalue}^\sharp$

This function maps each type to that type's default abstract value. It uses the number abstaction function α_n, the boolean abstraction function α_b, the string abstraction function α_{str}, and the reference abstraction function α_r.

$\texttt{defaultvalue}^\sharp \in \mathit{Type} \to \mathit{Value}^\sharp$

$\texttt{defaultvalue}^\sharp(\tau) =$

$$\begin{cases} \alpha_n(0) & \text{if } \tau = \texttt{int} \\ \alpha_b(\textbf{false}) & \text{if } \tau = \texttt{bool} \\ \alpha_{str}(\texttt{""}) & \text{if } \tau = \texttt{string} \\ \alpha_r(\textbf{null}) & \text{otherwise} \end{cases}$$

2.3.5 $\texttt{initstate}^\sharp$

This function takes the program and generates the initial abstract state. It is similar to the concrete version except that it uses the corresponding abstractions of the concrete values.

$\texttt{initstate}^\sharp \in \mathit{Program} \to \mathit{State}^\sharp$

$\texttt{initstate}^\sharp(p) = \left(\theta, \bullet, \hat{\rho}, \hat{\sigma}, \vec{\hat{k}}\right)$ where

$\theta = \texttt{foldl}(\ (\mathit{acc}, \mathit{class} \Rightarrow \mathit{acc} \cup [\mathit{class}.cn_1 \mapsto \texttt{initclass}^\sharp(\mathit{class})]),\ [\texttt{TopClass} \mapsto (\emptyset, \emptyset)],\ p)$

cn is the name of the first class in p

$\hat{a}$ is a fresh abstract address

$\hat{o} = (cn, \pi_1(\theta(cn)))$

$\hat{\sigma} = [\hat{a} \mapsto \hat{o}]$

$\mathit{methods} = \pi_2(\theta(cn))$

$\mathit{methods}(cn) = \textbf{def}\ cn(\overrightarrow{x : \tau}) : \tau_{ret}\ \{\ \vec{s} \cdot \textbf{return}\ \texttt{self}\ \}$

$\vec{\hat{k}} = \texttt{toSK}^\sharp(\vec{s})$

$\hat{\rho} = [\texttt{self} \mapsto \hat{a}] \cup [\ x_i \mapsto \texttt{defaultvalue}^\sharp(\tau_i)\ \mid\ 0 \le i < |\overrightarrow{x : \tau}|\ ]$

$\texttt{initclass}^\sharp \in \mathit{ClassDefs} \times \mathit{Class} \to (\mathit{Variable} \to \mathit{Value}^\sharp) \times (\mathit{MethodName} \to \mathit{Method})$

$\texttt{initclass}^\sharp(\theta, \mathit{class}) = (\mathit{fields}, \mathit{methods})$ where

$\mathit{class} = \textbf{class}\ cn_1\ \textbf{extends}\ cn_2\ \{\ \textbf{fields}\ \overrightarrow{x : \tau}\ \cdot\ \textbf{methods}\ \overrightarrow{m}\ \}$

$\mathit{superflds} = \pi_1(\theta(cn_2))$

$\mathit{supermethods} = \pi_2(\theta(cn_2))$

$\mathit{localflds} = [\ x_i \mapsto \tau_i\ \mid\ 0 \le i < |\overrightarrow{x : \tau}|\ ]$

$\mathit{localmethods} = [\ m_i.mn \mapsto m_i\ \mid\ 0 \le i < |\overrightarrow{m}|\ ]$

$\mathit{fields} = \mathit{superflds}[\mathit{localflds}]$

$\mathit{methods} = \mathit{supermethods}[\mathit{localmethods}]$

2.3.6 $\texttt{lookup}^\sharp$

This function takes a set of object addresses, an object field, and a heap, and returns the join of all the abstract values of that field in the objects at those addresses.

$\texttt{lookup}^\sharp \in \mathcal{P}(\mathit{Address}^\sharp) \times \mathit{Variable} \times \mathit{Heap}^\sharp \to \mathit{Value}^\sharp$

$\texttt{lookup}^\sharp\left(\overline{\hat{a}}, x, \hat{\sigma}\right) = \bigsqcup \left\{\ \hat{v}\ \mid\ \hat{a} \in \overline{\hat{a}},\ \pi_2(\hat{\sigma}(\hat{a}))(x) = \hat{v}\ \right\}$

2.3.7 $\texttt{toSK}^\sharp$

This function maps a sequence of statements to a sequence of $\widehat{\textbf{stmtK}}$ continuations containing those statements.

$\texttt{toSK}^\sharp \in \mathit{Stmt}^\star \to \mathit{Kont}^{\sharp\star}$

$\texttt{toSK}^\sharp(\vec{s}) = \vec{\hat{\kappa}}$ where $\hat{\kappa}_i = \widehat{\textbf{stmtK}}(s_i)$ for $0 \le i < |\vec{s}|$

2.3.8 $\texttt{update}^\sharp$

This function takes a set of object addresses, an object field, the new abstract value for that field, and a heap and returns an updated heap that has suitably updated the objects at those addresses. It performs a *strong* or *weak* update of those objects depending on if the given abstract addresses map to a single concrete address or not. The γ_a operator is the abstract address concretization operator; note that the *implementation* of this helper function *should not* actually use the concretization operator (which would be uncomputable).

$\texttt{update}^\sharp \in \mathit{Heap}^\sharp \times \mathcal{P}(\mathit{Address}^\sharp) \times \mathit{Variable} \times \mathit{Value}^\sharp \to \mathit{Heap}^\sharp$

$\texttt{update}^\sharp\left(\hat{\sigma}, \overline{\hat{a}}, x, \hat{v}\right) =$

$$\begin{cases} \hat{\sigma}[\hat{a} \mapsto \hat{o}[x \mapsto \hat{v}]] & \text{if } \overline{\hat{a}} = \{\hat{a}\},\ |\gamma_a(\hat{a})| = 1,\ \hat{\sigma}(\hat{a}) = \hat{o} \\ \hat{\sigma}[\hat{a} \mapsto \hat{o}[x \mapsto \hat{v} \sqcup \pi_2(\hat{o})(x)]] & \text{otherwise, for } \hat{a} \in \overline{\hat{a}},\ \hat{o} = \hat{\sigma}(\hat{a}) \end{cases}$$