Lwnn Concrete and Abstract Semantics

CS 260, Fall 2013

1 Lwnn Abstract Syntax

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

$$\begin{aligned}
p \in Program &::= \overrightarrow{class} \\
class \in Class &::= \textbf{class}\ cn_1\ \textbf{extends}\ cn_2\ \left\{\ \textbf{fields}\ \overrightarrow{x : \tau}\ \cdot\ \textbf{methods}\ \overrightarrow{m}\ \right\} \\
\tau \in Type &::= \texttt{int} \mid \texttt{bool} \mid \texttt{string} \mid \texttt{null} \mid cn \\
m \in Method &::= \textbf{def}\ mn(\overrightarrow{x : \tau}) : \tau_{ret}\ \{\ \vec{s} \cdot \textbf{return}\ e\ \} \\
s \in Stmt &::= x := e \mid e_1.x := e_2 \mid x := e.mn(\vec{e}) \mid x := \textbf{new}\ cn(\vec{e}) \\
&\quad \mid \textbf{if}\ e\ \overrightarrow{s_1}\ \textbf{else}\ \overrightarrow{s_2} \mid \textbf{while}\ e\ \vec{s} \\
e \in Exp &::= \overline{n} \mid \overline{b} \mid \overline{str} \mid \textbf{null} \mid x \mid e.x \mid e_1 \oplus e_2 \\
\oplus \in BinaryOp &::= + \mid - \mid \times \mid \div \mid < \mid \leq \mid \wedge \mid \vee \mid = \mid \neq
\end{aligned}$$

Notation. By abuse of notation we use the vector notation $\vec{\cdot}$ to indicate an ordered sequence of unspecified size n, indexed from $0 \leq i < n$. We use the overline notation $\bar{\cdot}$ to indicate an unordered set. The length of a vector (respectively, set) is denoted by $|\vec{\cdot}|$ (respectively, $|\bar{\cdot}|$).

Syntax Summary. A *program* consists of a sequence of classes. A *class* specifies a name, a superclass, a set of fields (each field consisting of a variable and its type), and a set of methods. *Types* represent integers, booleans, strings, **null**, or one of the user-defined classes, respectively (i.e., the `null` type has a single value, called **null**). A *method* specifies a name, a set of parameters (each consisting of a variable and its type), a return type, and a body. The body of a method is a sequence of statements terminated by a **return** statement. A *statement* is an assignment, an object field update, an object method call, new object construction, a conditional, or a while loop. An *expression* is a set of integers, a set of booleans, a set of strings, the **null** value, a variable, an object field access, or a binary operation. We use sets of integers, booleans, and strings to allow for nondeterministic execution without needing to specify I/O for the language.

Type System. The language is statically typed, using a nominal type system with subtyping and recursive types. The `int`, `bool`, and `string` types are invariant. The `null` type is a subtype of all classes. All classes are subtypes of a builtin class called `TopClass` that has no fields or methods. Every type has a default value of that type, which is used to initialize object fields and any method parameters that don't receive an argument. The default value of `int` is 0; the default value of `bool` is **false**; the default value of `string` is `""`; the default value of `null` and all class types is **null**.

Assumptions. All methods implicitly have a parameter `self`, which contains the address of the object that the method was called on (i.e., the `this` parameter in C++ and Java). Method calls can provide fewer arguments than there are parameters; the extra parameters are given default values (this is how methods can declare local variables). All classes contain a constructor method, defined as a method with the same name as that class; this method is called whenever an object of that class is created using **new**. All constructors should end with **return** `self`. When a program is executed, it takes the first class in the program and calls its constructor as the entry point to the program. In the sequence of class definitions, a superclass must be defined before any class that inherits from it.

Concrete Syntax. The concrete syntax allows various shortcuts by making some parts of the syntax optional. If these optional parts are left out, the parser will fill them in with default syntax to match the required abstract syntax. In particular:

- A class doesn't need to declare any fields or methods; if they are left out then the corresponding part of the abstract syntax will be the empty set.
- A class doesn't need to specify a superclass using **extend**; if this is left out then in the abstract syntax the class will extend `TopClass`.
- A method doesn't need to end with a **return**; if this is left out the abstract syntax will use **return** `self`.
- If a method doesn't syntactically contain a **return**, it doesn't need to specify the method's return type. The return type will be inferred in the abstract syntax to be the method's containing class.
- A method call doesn't need to assign the result to a variable. If there is no assignment, the abstract syntax will assign the return value to a dummy variable.
- An **if** statement doesn't need to have an **else** clause. If it is left out, the abstract syntax will contain an **else** clause with a single statement that effectively is a no-op.

2 Lwnn Concrete Semantics

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

2.1 Concrete Semantic Domains

$$
\begin{aligned}
\varsigma \in \mathit{State} &= \mathit{ClassDefs} \times \mathit{Stmt}^? \times \mathit{Locals} \times \mathit{Heap} \times \mathit{Kont}^\star \\
\theta \in \mathit{ClassDefs} &= \mathit{ClassName} \to ((\mathit{Variable} \to \mathit{Value}) \times (\mathit{MethodName} \to \mathit{Method})) \\
\rho \in \mathit{Locals} &= \mathit{Variable} \to \mathit{Value} \\
\sigma \in \mathit{Heap} &= \mathit{Address} \to \mathit{Object} \\
r \in \mathit{Reference} &= \mathit{Address} \uplus \{\textbf{null}\} \\
v \in \mathit{Value} &= \mathbb{Z} \uplus \mathit{Bool} \uplus \mathit{String} \uplus \mathit{Reference} \\
a \in \mathit{Address} &= \mathbb{N} \\
o \in \mathit{Object} &= \mathit{ClassName} \times (\mathit{Variable} \to \mathit{Value}) \\
\kappa \in \mathit{Kont} &= \textbf{stmtK}\ \mathit{Stmt} \ \uplus\ \textbf{whileK}\ \mathit{Exp} \times \mathit{Stmt}^\star \ \uplus\ \textbf{retK}\ \mathit{Variable} \times \mathit{Exp} \times \mathit{Locals}
\end{aligned}
$$

Notation. We borrow notation from formal languages: $\cdot^?$ means 0 or 1 instances; $\cdot^\star$ means an ordered sequence of 0 or more instances; $\cdot^+$ means an ordered sequence of 1 or more instances. The $\uplus$ operator means disjoint union.

Domains Summary. A state consists of the class definitions (which are invariant across all states), an optional statement to be processed, a map from the current method's local variables to their values, a heap mapping addresses to objects, and a continuation stack. The class definitions map each class name to a pair of maps; the first maps the class's fields to their default values, and the second maps the class's method names to the method definitions. Language values are integers, booleans, strings, object references. An object reference is either an address or **null**. An object is a tuple of the object's class name and a map from the class field's to their values for this object. The continuation stack is a sequence of **stmtK** continuations (holding statements to be processed), **whileK** continuations (holding the guard and body of a currently executing while loop, so we can start the next iteration), and **retK** continuations (holding the variable to receive the callee method's return value, the expression whose value should be returned from the callee, and the caller method's local variables so that we can restore them when the callee returns).

2.2 Concrete Transition Rules

Table 1: The concrete transition relation. Each rule describes how to take one concrete state $(\theta, s^?, \rho, \sigma, \kappa \cdot \overrightarrow{\kappa_1})$ to the next concrete state $(\theta, s^?_{new}, \rho_{new}, \sigma_{new}, \overrightarrow{\kappa_{new}})$. 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 stack indicates appending sequences; thus κ is the top of the continuation stack in the source state and $\overrightarrow{\kappa_1}$ is the rest of that continuation stack.

no.	$s^?$	**premises**	$s^?_{new}$	ρ_{new}	σ_{new}	$\overrightarrow{\kappa_{new}}$
1	$x := e$	$[\![e]\!] = v$	$\bullet$	$\rho[x \mapsto v]$	σ	$\overrightarrow{\kappa_1}$
2	$e_1.x := e_2$	$[\![e_1]\!] = a,\ [\![e_2]\!] = v,\ o = \sigma(a)[x \mapsto v]$	$\bullet$	ρ	$\sigma[a \mapsto o]$	$\overrightarrow{\kappa_1}$
3	$x := e.mn(\vec{e})$	$\left(\rho_1, \overrightarrow{\kappa_2}\right) = \texttt{call}\left(\theta, x, [\![e]\!], \sigma, mn, \overrightarrow{[\![e]\!]}, \rho\right)$	$\bullet$	ρ_1	σ	$\overrightarrow{\kappa_2} \cdot \overrightarrow{\kappa_1}$
4	$x :=$ **new** $cn(\vec{e})$	$\left(\rho_1, \sigma_1, \overrightarrow{\kappa_2}\right) = \texttt{construct}\left(\theta, x, cn, \overrightarrow{[\![e]\!]}, \rho, \sigma\right)$	$\bullet$	ρ_1	σ_1	$\overrightarrow{\kappa_2} \cdot \overrightarrow{\kappa_1}$
5	**if** e $\overrightarrow{s_1}$ **else** $\overrightarrow{s_2}$	$[\![e]\!] =$ **true**	$\bullet$	ρ	σ	$\texttt{toSK}(\overrightarrow{s_1}) \cdot \overrightarrow{\kappa_1}$
6	**if** e $\overrightarrow{s_1}$ **else** $\overrightarrow{s_2}$	$[\![e]\!] =$ **false**	$\bullet$	ρ	σ	$\texttt{toSK}(\overrightarrow{s_2}) \cdot \overrightarrow{\kappa_1}$
7	**while** e $\vec{s}$	$[\![e]\!] =$ **true**	$\bullet$	ρ	σ	$\texttt{toSK}(\vec{s}) \cdot \kappa \cdot \overrightarrow{\kappa_1}$
8	**while** e $\vec{s}$	$[\![e]\!] =$ **false**	$\bullet$	ρ	σ	$\overrightarrow{\kappa_1}$
9	$\bullet$	$\kappa =$ **retK**(x, e, ρ_1), $[\![e]\!] = v$	$\bullet$	$\rho_1[x \mapsto v]$	σ	$\overrightarrow{\kappa_1}$
10	$\bullet$	$\kappa =$ **stmtK**(s_1)	s_1	ρ	σ	$\overrightarrow{\kappa_1}$
11	$\bullet$	$\kappa =$ **whileK**$(e, \vec{s})$, $[\![e]\!] =$ **true**	$\bullet$	ρ	σ	$\texttt{toSK}(\vec{s}) \cdot \kappa \cdot \overrightarrow{\kappa_1}$
12	$\bullet$	$\kappa =$ **whileK**$(e, \vec{s})$, $[\![e]\!] =$ **false**	$\bullet$	ρ	σ	$\overrightarrow{\kappa_1}$

Notation. We use $[\![e]\!]$ as shorthand for $\eta(e, \rho, \sigma)$ when ρ and σ are obvious from context. We use $s^?$ to indicate 0 or 1 statements; $\bullet$ means there is no statement. For any map X, the notation $X[a \mapsto b]$ means a new map that is exactly the same as X except that a maps to b. We abuse notation for objects by using this map update notation to update object fields in rule 2, even though technically objects are a pair of class name and a map. We use $\pi_i(tuple)$ to project out the ith element of $tuple$.

2.3 Concrete Helper Functions

We describe the helper functions used by the transition rules. The functions are listed in alphabetical order. Note that in several places we implictly assume that a reference value must be an address rather then **null**; this means that the behavior if the reference is actually **null** is undefined.

Notation. The notation $map_1[map_2]$ is shorthand for updating map_1 with each entry in map_2 in turn. Recall that we use $\pi_i(tuple)$ to project out the ith element of $tuple$.

2.3.1 $\eta(e, \rho, \sigma)$ **a.k.a.** $[\![e]\!]$

This function describes how to evaluate expressions to values. Note that sets of integers/booleans/strings are evaluated by nondeterministically selecting an element from that set. Variables are looked up in the locals map; object field access gets the address of an object and then looks up the given field's value in that object; binary operators recursively evaluate the operands and then apply the appropriate operation to the result (the operators are described below).

$\eta : Exp \times Locals \times Heap \to Value$
$\eta(e, \rho, \sigma) =$

$$\begin{cases} n & \text{if } e = \overline{n},\ n \in \overline{n} \\ b & \text{if } e = \overline{b},\ b \in \overline{b} \\ str & \text{if } e = \overline{str},\ str \in \overline{str} \\ \textbf{null} & \text{if } e = \textbf{null} \\ \rho(x) & \text{if } e = x \\ fields(x) & \text{if } e = e_1.x,\ [\![e_1]\!] = a,\ \pi_2(\sigma(a)) = fields \\ [\![e_1]\!] \oplus [\![e_2]\!] & \text{if } e = e_1 \oplus e_2 \end{cases}$$

Operators on Integers. $\{+, -, \times, \div\} : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ are the standard (unbounded width) integer arithmetic operators.

Operators on Booleans. $\{\wedge, \vee\} : Bool \times Bool \to Bool$ are the standard logical AND and OR operators.

Operators and Strings. $+ : String \times String \to String$ is string concatenation. $\{<, \leq\} : String \times String \to Bool$ are strict and reflexive lexicographic string comparison, respectively.

Operators on Values. $\{=, \neq\} : Value \times Value \to Bool$ are equality and inequality of values, respectively.

2.3.2 `call`

This function describes how to process a method call. Note that the current locals map is saved in the **retK** continuation to be restored once the callee returns, and that `self` is mapped to the object's address in the new locals map. Any arguments are copied to their respective parameters; any parameters not given an argument are mapped to their type's default value.

$\texttt{call} \in ClassDefs \times Variable \times Address \times Heap \times MethodName \times Value^\star \times Locals \to Locals \times Kont^\star$

$\texttt{call}\,(\theta, x, a, \sigma, mn, \vec{v}, \rho) = \left(\rho_1, \overrightarrow{\kappa_1}\right)$ where

$cn = \pi_1(\sigma(a))$

$methods = \pi_2(\theta(cn))$

$methods(mn) = \textbf{def}\ mn(\overrightarrow{x : \tau}) : \tau_{ret}\ \{\ \vec{s} \cdot \textbf{return}\ e\ \}$

$\overrightarrow{\kappa_1} = \texttt{toSK}(\vec{s}) \cdot \textbf{retK}(x,\ e,\ \rho)$

$\rho_1 = [\texttt{self} \mapsto a] \cup [\ x_i \mapsto v\ \mid\ 0 \leq i < |\vec{v}| \implies v = v_i,\ |\vec{v}| \leq i < |\overrightarrow{x : \tau}| \implies v = \texttt{defaultvalue}(\tau_i)\ ]$

2.3.3 `construct`

This function describes how to create a new object. It retrieves the class's fields and their default values from the class definitions to create a new object, then allocates a fresh address and creates a new heap that maps the address to the new object. It then retrieves the constructor method for that class and proceeds as if for a method call. Note that since constructors must end in **return** `self`, x will get the new object's address when the constructor returns.

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

$\texttt{construct}\,(\theta, x, mn, \vec{v}, \rho, \sigma) = \left(\rho_1, \sigma_1, \overrightarrow{\kappa_1}\right)$ where

a is a fresh address

$o = (cn, \pi_1(\theta(cn)))$

$\sigma_1 = \sigma[a \mapsto o]$

$methods = \pi_2(\theta(cn))$

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

$\overrightarrow{\kappa_1} = \texttt{toSK}(\vec{s}) \cdot \textbf{retK}(x, \texttt{self}, \rho)$

$\rho_1 = [\texttt{self} \mapsto a] \cup [\ x_i \mapsto v\ \mid\ 0 \le i < |\vec{v}| \implies v = v_i,\ |\vec{v}| \le i < |\overrightarrow{x : \tau}| \implies v = \texttt{defaultvalue}(\tau_i)\ ]$

2.3.4 `defaultvalue`

This function maps each type to that type's default value.

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

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

$$\begin{cases} 0 & \text{if } \tau = \texttt{int} \\ \textbf{false} & \text{if } \tau = \texttt{bool} \\ \texttt{""} & \text{if } \tau = \texttt{string} \\ \textbf{null} & \text{otherwise} \end{cases}$$

2.3.5 `initstate`

This function takes the program and generates the initial state. It creates the class definitions and calls the constructor of the first class in the program as the starting point of the program's execution. It uses a secondary helper function `initclass` to convert a syntactic class definition into a semantic class definition. `foldl` is the standard functional fold-left function that takes a function, an initial value, and a sequence and applies the function to each element of the sequence, passing the result of each function call to the next call in the chain.

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

$\texttt{initstate}(p) = (\theta, \bullet, \rho, \sigma, \vec{\kappa})$ where

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

cn is the name of the first class in p

a is a fresh address

$o = (cn, \pi_1(\theta(cn)))$

$\sigma = [a \mapsto o]$

$methods = \pi_2(\theta(cn))$

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

$\vec{\kappa} = \texttt{toSK}(\vec{s})$

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

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

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

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

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

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

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

$\mathit{localmethods} = [\ m_j.mn \mapsto m_j\ \mid\ 0 \le j < |\vec{m}|\ ]$

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

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

2.3.6 toSK

This function maps a sequence of statements to a sequence of **stmtK** continuations containing those statements.

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

$\texttt{toSK}(\vec{s}) = \vec{\kappa}$ where $\kappa_i = \textbf{stmtK}(s_i)$ for $0 \le i < |\vec{s}|$

3 Lwnn Abstract Semantics

We describe the semantic domains that constitute an abstract state (Section 3.1), abstract state transition rules (Section 3.2), and the helper functions used by the abstract transition rules (Section 3.3).

3.1 Abstract Semantic Domains

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

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

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 $\mathit{Locals}^\sharp$ domain to include a set 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**.

3.2 Abstract Transition Rules

Table 2: The abstract transition relation. Each rule describes how to take one abstract state $(\hat{\theta}, s^?, \hat{\rho}, \hat{\sigma}, \hat{\kappa} \cdot \overrightarrow{\hat{\kappa}_1})$ to the next abstract state $(\hat{\theta}, s^?_{new}, \hat{\rho}_{new}, \hat{\sigma}_{new}, \overrightarrow{\hat{\kappa}_{new}})$. 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	$x := e$	$[\![e]\!]^\sharp = \hat{v}$	$\bullet$	$\hat{\rho}[x \mapsto \hat{v}]$	$\hat{\sigma}$	$\overrightarrow{\hat{\kappa}_1}$
2	$e_1.x := e_2$	$\hat{\sigma}_1 = \texttt{update}^\sharp(\hat{\sigma}, [\![e_1]\!]^\sharp, x, [\![e_2]\!]^\sharp)$	$\bullet$	$\hat{\rho}$	$\hat{\sigma}_1$	$\overrightarrow{\hat{\kappa}_1}$
3	$x := e.mn(\vec{e})$	$\left(\hat{\rho}_1, \overrightarrow{\hat{\kappa}_2}\right) \in \texttt{call}^\sharp\left(\hat{\theta}, x, [\![e]\!]^\sharp, \hat{\sigma}, mn, \overrightarrow{[\![e]\!]^\sharp}, \hat{\rho}, \overrightarrow{\hat{\kappa}_1}\right)$	$\bullet$	$\hat{\rho}_1$	$\hat{\sigma}$	$\overrightarrow{\hat{\kappa}_2}$
4	$x :=$ **new** $cn(\vec{e})$	$\left(\hat{\rho}_1, \hat{\sigma}_1, \overrightarrow{\hat{\kappa}_2}\right) = \texttt{construct}^\sharp\left(\hat{\theta}, x, cn, \overrightarrow{[\![e]\!]^\sharp}, \hat{\rho}, \hat{\sigma}, \overrightarrow{\hat{\kappa}_1}\right)$	$\bullet$	$\hat{\rho}_1$	$\hat{\sigma}_1$	$\overrightarrow{\hat{\kappa}_2}$
5	**if** e $\overrightarrow{s_1}$ **else** $\overrightarrow{s_2}$	$\textbf{true} \in \gamma_b([\![e]\!]^\sharp)$	$\bullet$	$\hat{\rho}$	$\hat{\sigma}$	$\texttt{toSK}^\sharp(\overrightarrow{s_1}) \cdot \overrightarrow{\hat{\kappa}_1}$
6	**if** e $\overrightarrow{s_1}$ **else** $\overrightarrow{s_2}$	$\textbf{false} \in \gamma_b([\![e]\!]^\sharp)$	$\bullet$	$\hat{\rho}$	$\hat{\sigma}$	$\texttt{toSK}^\sharp(\overrightarrow{s_2}) \cdot \overrightarrow{\hat{\kappa}_1}$
7	**while** e $\vec{s}$	$\textbf{true} \in \gamma_b([\![e]\!]^\sharp)$	$\bullet$	$\hat{\rho}$	$\hat{\sigma}$	$\texttt{toSK}^\sharp(\vec{s}) \cdot \hat{\kappa} \cdot \overrightarrow{\hat{\kappa}_1}$
8	**while** e $\vec{s}$	$\textbf{false} \in \gamma_b([\![e]\!]^\sharp)$	$\bullet$	$\hat{\rho}$	$\hat{\sigma}$	$\overrightarrow{\hat{\kappa}_1}$
9	$\bullet$	$\hat{\kappa} = \widehat{\textbf{finK}}, \widehat{\textbf{retK}}(x, e, \hat{\rho}_1) \cdot \overrightarrow{\hat{\kappa}_2} \in \pi_2(\hat{\rho}),\ [\![e]\!]^\sharp = \hat{v}$	$\bullet$	$\hat{\rho}_1[x \mapsto \hat{v}]$	$\hat{\sigma}$	$\overrightarrow{\hat{\kappa}_2}$
10	$\bullet$	$\hat{\kappa} = \widehat{\textbf{stmtK}}(s_1)$	s_1	$\hat{\rho}$	$\hat{\sigma}$	$\overrightarrow{\hat{\kappa}_1}$
11	$\bullet$	$\hat{\kappa} = \widehat{\textbf{whileK}}(e, \vec{s}),\ \textbf{true} \in \gamma_b([\![e]\!]^\sharp)$	$\bullet$	$\hat{\rho}$	$\hat{\sigma}$	$\texttt{toSK}^\sharp(\vec{s}) \cdot \hat{\kappa} \cdot \overrightarrow{\hat{\kappa}_1}$
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 use the indirection provided by $Locals^\sharp$ to store the current continuation stack inside the continuation stack set of $\hat{\rho}_1$ for a method/constructor call, and rule 9 restores the continuation stack from that set when returning from the callee.

3.3 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.

3.3.1 $\eta^\sharp(e, \hat{\rho}, \hat{\sigma})$ **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.

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

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

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

3.3.2 $\texttt{call}^\sharp$

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

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

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

$\overline{(\hat{a}, cn)} = \left\{ (\hat{a},\ \pi_1(\sigma(\hat{a}))) \mid \hat{a} \in \overline{\hat{a}} \right\}$

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

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

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

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

3.3.3 $\texttt{construct}^\sharp$

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).

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

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

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

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

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

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

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

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

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

3.3.4 `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.

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

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

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

3.3.5 `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.

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

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

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

cn is the name of the first class in p

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

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

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

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

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

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

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

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

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

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

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

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

$\mathit{localflds} = [\ x_i \mapsto \texttt{defaultvalue}^\sharp(\tau_i)\ \mid\ 0 \le i < |\overrightarrow{x : \tau}|\ ]$

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

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

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

3.3.6 $\texttt{lookup}^\sharp$

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.

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

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

3.3.7 $\texttt{toSK}^\sharp$

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 $0 \le i < |\vec{s}|$

3.3.8 $\texttt{update}^\sharp$

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).

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

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

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