

Lwnn Concrete Semantics

CS 260

1 Lwnn Abstract Syntax

$$\begin{aligned} n &\in \mathbb{Z} & b &\in \text{Bool} & str &\in \text{String} & x &\in \text{Variable} \\ cn &\in \text{ClassName} & mn &\in \text{MethodName} \\ p \in \text{Program} &::= \overrightarrow{\text{class}} \\ class \in \text{Class} &::= \mathbf{class} \, cn_1 \, \mathbf{extends} \, cn_2 \, \{ \mathbf{fields} \, \overline{x} : \overline{\tau} \cdot \mathbf{methods} \, \overrightarrow{m} \} \\ \tau \in \text{Type} &::= \text{int} \mid \text{bool} \mid \text{string} \mid \text{null} \mid cn \\ m \in \text{Method} &::= \mathbf{def} \, mn(\overline{x} : \overline{\tau}) : \tau_{ret} \{ \overrightarrow{s} \cdot \mathbf{return} \, e \} \\ s \in \text{Stmt} &::= x := e \mid e_1.x := e_2 \mid x := e.mn(\overrightarrow{e}) \mid x := \mathbf{new} \, cn(\overrightarrow{e}) \\ &\mid \mathbf{if} \, e \, \overrightarrow{s}_1 \, \mathbf{else} \, \overrightarrow{s}_2 \mid \mathbf{while} \, e \, \overrightarrow{s} \\ e \in \text{Exp} &::= \overline{n} \mid \overline{b} \mid \overline{str} \mid \mathbf{null} \mid x \mid e.x \mid e_1 \oplus e_2 \\ \oplus \in \text{BinaryOp} &::= + \mid - \mid \times \mid \div \mid < \mid \leq \mid \wedge \mid \vee \mid = \mid \neq \end{aligned}$$

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

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. We require that the first parameter of every method is named *self*, which will be a pointer to the receiver object of the method call. We assume that *self* is immutable, i.e., it cannot be assigned to. 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 have an explicit *self* parameter, it will be added by the parser to get the correct abstract syntax.
- 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

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

Notation. We borrow notation from formal languages: $\cdot^?$ means 0 or 1 instances; \cdot^* 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 types, 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 $(\theta, s^?, \rho, \sigma, \vec{\kappa})$ to the next concrete state $(\theta, s_{new}^?, \rho_{new}, \sigma_{new}, \vec{\kappa}_{new})$, where $\vec{\kappa} = \kappa \cdot \vec{\kappa}_1$. 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 stack indicates appending sequences; thus κ is the top of the continuation stack in the source state and $\vec{\kappa}_1$ is the rest of that continuation stack.

no.	$s^?$	premises	$s_{new}^?$	ρ_{new}	σ_{new}	$\vec{\kappa}_{new}$
1	$x := e$	$\llbracket e \rrbracket = v$	\bullet	$\rho[x \mapsto v]$	σ	$\vec{\kappa}$
2	$e_1.x := e_2$	$\llbracket e_1 \rrbracket = a, \llbracket e_2 \rrbracket = v, o = \sigma(a)[x \mapsto v]$	\bullet	ρ	$\sigma[a \mapsto o]$	$\vec{\kappa}$
3	$x := e.mn(\vec{e})$	$(\rho_1, \vec{\kappa}_2) = \text{call}(\theta, x, \llbracket e \rrbracket, \sigma, mn, \llbracket \vec{e} \rrbracket, \rho)$	\bullet	ρ_1	σ	$\vec{\kappa}_2 \cdot \vec{\kappa}$
4	$x := \mathbf{new} \text{ } cn(\vec{e})$	$(\rho_1, \sigma_1, \vec{\kappa}_2) = \text{construct}(\theta, x, cn, \llbracket \vec{e} \rrbracket, \rho, \sigma)$	\bullet	ρ_1	σ_1	$\vec{\kappa}_2 \cdot \vec{\kappa}$
5	if $e \xrightarrow{s_1}$ else \vec{s}_2	$\llbracket e \rrbracket = \mathbf{true}$	\bullet	ρ	σ	$\text{toSK}(\vec{s}_1) \cdot \vec{\kappa}$
6	if $e \xrightarrow{s_1}$ else \vec{s}_2	$\llbracket e \rrbracket = \mathbf{false}$	\bullet	ρ	σ	$\text{toSK}(\vec{s}_2) \cdot \vec{\kappa}$
7	while $e \vec{s}$	$\llbracket e \rrbracket = \mathbf{true}$	\bullet	ρ	σ	$\text{toSK}(\vec{s}) \cdot \mathbf{whileK}(e, \vec{s}) \cdot \vec{\kappa}$
8	while $e \vec{s}$	$\llbracket e \rrbracket = \mathbf{false}$	\bullet	ρ	σ	$\vec{\kappa}$
9	\bullet	$\kappa = \mathbf{retK}(x, e, \rho_1), \llbracket e \rrbracket = v$	\bullet	$\rho_1[x \mapsto v]$	σ	$\vec{\kappa}_1$
10	\bullet	$\kappa = \mathbf{stmtK}(s_1)$	s_1	ρ	σ	$\vec{\kappa}_1$
11	\bullet	$\kappa = \mathbf{whileK}(e, \vec{s}), \llbracket e \rrbracket = \mathbf{true}$	\bullet	ρ	σ	$\text{toSK}(\vec{s}) \cdot \kappa \cdot \vec{\kappa}_1$
12	\bullet	$\kappa = \mathbf{whileK}(e, \vec{s}), \llbracket e \rrbracket = \mathbf{false}$	\bullet	ρ	σ	$\vec{\kappa}_1$

Notation. We use $\llbracket e \rrbracket$ 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 $\pi_i(\text{tuple})$ to project out the i th 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 implicitly assume that a reference value must be an address rather than **null**; this means that the behavior if the reference is actually **null** is undefined.

Notation. The notation $\text{map}_1[\text{map}_2]$ is shorthand for updating map_1 with each entry in map_2 in turn. Recall that we use $\pi_i(\text{tuple})$ to project out the i th element of tuple .

2.3.1 $\eta(e, \rho, \sigma)$ a.k.a. $\llbracket e \rrbracket$

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).

$\eta : \text{Exp} \times \text{Locals} \times \text{Heap} \rightarrow \text{Value}$

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

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

Operators on Integers. $\{+, -, \times, \div\} : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$ and $\{<, \leq\} : \mathbb{Z} \times \mathbb{Z} \rightarrow \text{Bool}$ are the standard (unbounded width) integer arithmetic operators.

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

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

Operators on Values. $\{=, \neq\} : \text{Value} \times \text{Value} \rightarrow \text{Bool}$ 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.

$\text{call} \in \text{ClassDefs} \times \text{Variable} \times \text{Address} \times \text{Heap} \times \text{MethodName} \times \text{Value}^* \times \text{Locals} \rightarrow \text{Locals} \times \text{Kont}^*$

$\text{call}(\theta, x, a, \sigma, mn, \vec{v}, \rho) = (\rho_1, \vec{\kappa}_1)$ where

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

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

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

$\vec{\kappa}_1 = \text{toSK}(\vec{s}) \cdot \text{retK}(x, e, \rho)$

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

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.

$\text{construct} \in \text{ClassDefs} \times \text{Variable} \times \text{ClassName} \times \text{Value}^* \times \text{Locals} \times \text{Heap} \rightarrow \text{Locals} \times \text{Heap} \times \text{Kont}^*$

$\text{construct}(\theta, x, mn, \vec{v}, \rho, \sigma) = (\rho_1, \sigma_1, \vec{\kappa}_1)$ where

a is a fresh address

$flds = [x \mapsto v \mid \pi_1(\theta(cn))(x) = \tau, \text{defaultvalue}(\tau) = v]$

$o = (cn, flds)$

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

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

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

$\vec{\kappa}_1 = \text{toSK}(\vec{s}) \cdot \text{retK}(x, \text{self}, \rho)$

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

2.3.4 defaultvalue

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

$\text{defaultvalue} \in \text{Type} \rightarrow \text{Value}$

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

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

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.

$\text{initstate} \in \text{Program} \rightarrow \text{State}$

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

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

cn is the name of the first class in p

a is a fresh address

$\text{flds} = [x \mapsto v \mid \pi_1(\theta(cn))(x) = \tau, \text{defaultvalue}(\tau) = v]$

$o = (cn, \text{flds})$

$\sigma = [a \mapsto o]$

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

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

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

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

$\text{initclass} \in \text{ClassDefs} \times \text{Class} \rightarrow (\text{Variable} \rightarrow \text{Type}) \times (\text{MethodName} \rightarrow \text{Method})$

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

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

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

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

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

$\text{localmethods} = [m_j.mn \mapsto m_j \mid 0 \leq j < |\overrightarrow{m}|]$

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

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

2.3.6 toSK

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

$\text{toSK} \in \text{Stmt}^* \rightarrow \text{Kont}^*$

$\text{toSK}(\vec{s}) = \vec{\kappa}$ where $\kappa_i = \mathbf{stmtK}(s_i)$ for $0 \leq i < |\vec{s}|$