

Operational Semantics of Cool

Lecture 16

Prof. Necula CS 164

1

Lecture Outline

- COOL operational semantics
- Motivation
- Notation
- The rules

Prof. Necula CS 164

2

Motivation

- We must specify for every Cool expression what happens when it is evaluated
 - This is the "meaning" of an expression
- The definition of a programming language:
 - The tokens \Rightarrow lexical analysis
 - The grammar \Rightarrow syntactic analysis
 - The typing rules \Rightarrow semantic analysis
 - The evaluation rules
 - \Rightarrow code generation and optimization

Prof. Necula CS 164

3

Evaluation Rules So Far

- So far, we specified the evaluation rules indirectly
 - We specified the compilation of Cool to a stack machine
 - And we specified the evaluation rules of the stack machine
- This is a complete description
- Why isn't it good enough?

Prof. Necula CS 164

4

Assembly Language Description of Semantics

- Assembly-language descriptions of language implementation have too many irrelevant details
 - Whether to use a stack machine or not
 - Which way the stack grows
 - How integers are represented on a particular machine
 - The particular instruction set of the architecture
- We need a complete but not overly restrictive specification

Prof. Necula CS 164

5

Programming Language Semantics

- There are many ways to specify programming language semantics
- They are all equivalent but some are more suitable to various tasks than others
- Operational semantics
 - Describes the evaluation of programs on an abstract machine
 - Most useful for specifying implementations
 - This is what we will use for Cool

Prof. Necula CS 164

6

Other Kinds of Semantics

- Denotational semantics
 - The meaning of a program is expressed as a mathematical object
 - Elegant but quite complicated
- Axiomatic semantics
 - Useful for checking that programs satisfy certain correctness properties
 - e.g., that the quick sort function sorts an array
 - The foundation of many program verification systems

Prof. Necula CS 164

7

Introduction to Operational Semantics

- Once, again we introduce a formal notation
 - Using logical rules of inference, just like for typing
- Recall the typing judgment
$$\text{Context} \vdash e : C$$
(in the given **context**, expression **e** has type **C**)
- We try something similar for evaluation
$$\text{Context} \vdash e : v$$
(in the given **context**, expression **e** evaluates to value **v**)

Prof. Necula CS 164

8

Example of Inference Rule for Operational Semantics

- Example:

$$\frac{\begin{array}{l} \text{Context} \vdash e_1 : 5 \\ \text{Context} \vdash e_2 : 7 \end{array}}{\text{Context} \vdash e_1 + e_2 : 12}$$

- In general the result of evaluating an expression depends on the result of evaluating its subexpressions
- The logical rules specify everything that is needed to evaluate an expression

What Contexts Are Needed?

- Obs.: Contexts are needed to handle variables
- Consider the evaluation of $y \leftarrow x + 1$
 - We need to keep track of values of variables
 - We need to allow variables to change their values during the evaluation
- We track variables and their values with:
 - An environment : tells us at what address in memory is the value of a variable stored
 - A store : tells us what is the contents of a memory location

Variable Environments

- A variable environment is a map from variable names to locations
- Tells in what memory location the value of a variable is stored
- Keeps track of which variables are in scope
- Example:

$$E = [a : l_1, b : l_2]$$

- To lookup a variable a in environment E we write $E(a)$

Stores

- A store maps memory locations to values
- Example:

$$S = [l_1 \rightarrow 5, l_2 \rightarrow 7]$$

- To lookup the contents of a location l_1 in store S we write $S(l_1)$
- To perform an assignment of 12 to location l_1 we write $S[12/l_1]$
 - This denotes a store S' such that

$$S'(l_1) = 12 \quad \text{and} \quad S'(l) = S(l) \text{ if } l \neq l_1$$

Cool Values

- All values in Cool are objects
 - All objects are instances of some class (the dynamic type of the object)
- To denote a Cool object we use the notation $X(a_1 = l_1, \dots, a_n = l_n)$ where
 - X is the dynamic type of the object
 - a_i are the attributes (including those inherited)
 - l_i are the locations where the values of attributes are stored

Cool Values (Cont.)

- Special cases (classes without attributes)
 - $Int(5)$ the integer 5
 - $Bool(true)$ the boolean true
 - $String(4, "Cool")$ the string "Cool" of length 4
- There is a special value `void` that is a member of all types
 - No operations can be performed on it
 - Except for the test `isvoid`
 - Concrete implementations might use NULL here

Operational Rules of Cool

- The evaluation judgment is

$$so, E, S \vdash e : v, S'$$

read:

- Given so the current value of the *self* object
- And E the current variable environment
- And S the current store
- If the evaluation of e terminates then
- The returned value is v
- And the new store is S'

Notes

- The "result" of evaluating an expression is a value and a new store
- Changes to the store model the side-effects
- The variable environment does not change
- Nor does the value of "self"
- The operational semantics allows for non-terminating evaluations
- We define one rule for each kind of expression

Operational Semantics for Base Values

$so, E, S \vdash \text{true} : \text{Bool}(\text{true}), S$

$so, E, S \vdash \text{false} : \text{Bool}(\text{false}), S$

i is an integer literal

$so, E, S \vdash i : \text{Int}(i), S$

s is a string literal

n is the length of s

$so, E, S \vdash s : \text{String}(n,s), S$

- No side effects in these cases
(the store does not change)

Operational Semantics of Variable References

$$\frac{\begin{array}{l} E(\text{id}) = l_{\text{id}} \\ S(l_{\text{id}}) = v \end{array}}{so, E, S \vdash \text{id} : v, S}$$

- Note the double lookup of variables
 - First from name to location
 - Then from location to value
- The store does not change
- A special case:

$so, E, S \vdash \text{self} : so, S$

Operational Semantics of Assignment

$$\frac{\begin{array}{l} \text{so, E, S} \vdash e : v, S_1 \\ E(\text{id}) = l_{\text{id}} \\ S_2 = S_1[v/l_{\text{id}}] \end{array}}{\text{so, E, S} \vdash \text{id} \leftarrow e : v, S_2}$$

- A three step process
 - Evaluate the right hand side
 - ⇒ a value and a new store S_1
 - Fetch the location of the assigned variable
 - The result is the value v and an updated store
- The environment does not change

Prof. Necula CS 164

19

Operational Semantics of Conditionals

$$\frac{\begin{array}{l} \text{so, E, S} \vdash e_1 : \text{Bool}(\text{true}), S_1 \\ \text{so, E, S}_1 \vdash e_2 : v, S_2 \end{array}}{\text{so, E, S} \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : v, S_2}$$

- The “threading” of the store enforces an evaluation sequence
 - e_1 must be evaluated first to produce S_1
 - Then e_2 can be evaluated
- The result of evaluating e_1 is a boolean object
 - The typing rules ensure this
 - There is another, similar, rule for $\text{Bool}(\text{false})$

Prof. Necula CS 164

20

Operational Semantics of Sequences

$$\frac{\begin{array}{l} \text{so, E, S} \vdash e_1 : v_1, S_1 \\ \text{so, E, S}_1 \vdash e_2 : v_2, S_2 \\ \dots \\ \text{so, E, S}_{n-1} \vdash e_n : v_n, S_n \end{array}}{\text{so, E, S} \vdash \{ e_1; \dots; e_n \} : v_n, S_n}$$

- Again the threading of the store expresses the intended evaluation sequence
- Only the last value is used
- But all the side-effects are collected

Prof. Neula CS 164

21

Operational Semantics of **while** (I)

$$\frac{\text{so, E, S} \vdash e_1 : \text{Bool}(\text{false}), S_1}{\text{so, E, S} \vdash \text{while } e_1 \text{ loop } e_2 \text{ pool} : \text{void}, S_1}$$

- If e_1 evaluates to **Bool(false)** then the loop terminates immediately
 - With the side-effects from the evaluation of e_1
 - And with result value **void**
- The typing rules ensure that e_1 evaluates to a boolean object

Prof. Neula CS 164

22

Operational Semantics of **while** (II)

$$\frac{\begin{array}{l} \text{so, } E, S \vdash e_1 : \text{Bool}(\text{true}), S_1 \\ \text{so, } E, S_1 \vdash e_2 : v, S_2 \\ \text{so, } E, S_2 \vdash \text{while } e_1 \text{ loop } e_2 \text{ pool} : \text{void}, S_3 \end{array}}{\text{so, } E, S \vdash \text{while } e_1 \text{ loop } e_2 \text{ pool} : \text{void}, S_3}$$

- Note the sequencing ($S \rightarrow S_1 \rightarrow S_2 \rightarrow S_3$)
- Note how looping is expressed
 - Evaluation of "**while ...**" is expressed in terms of the evaluation of itself in another state
- The result of evaluating e_2 is discarded
 - Only the side-effect is preserved

Prof. Necula CS 164

23

Operational Semantics of **let** Expressions (I)

$$\frac{\begin{array}{l} \text{so, } E, S \vdash e_1 : v_1, S_1 \\ \text{so, } ?, ? \vdash e_2 : v, S_2 \end{array}}{\text{so, } E, S \vdash \text{let } id : T \leftarrow e_1 \text{ in } e_2 : v_2, S_2}$$

- What is the context in which e_2 must be evaluated?
 - Environment like E but with a new binding of id to a fresh location l_{new}
 - Store like S_1 but with l_{new} mapped to v_1

Prof. Necula CS 164

24

Operational Semantics of **let** Expressions (II)

- We write $l_{\text{new}} = \text{newloc}(S)$ to say that l_{new} is a location that is not already used in S
 - Think of **newloc** as the dynamic memory allocation function
- The operational rule for **let**:

$$\frac{\begin{array}{l} s_0, E, S \vdash e_1 : v_1, S_1 \\ l_{\text{new}} = \text{newloc}(S_1) \\ s_0, E[l_{\text{new}}/\text{id}], S_1[v_1/l_{\text{new}}] \vdash e_2 : v_2, S_2 \end{array}}{s_0, E, S \vdash \text{let id} : T \leftarrow e_1 \text{ in } e_2 : v_2, S_2}$$

Prof. Neula CS 164

25

Operational Semantics of **new**

- Consider the expression **new T**
- Informal semantics
 - Allocate new locations to hold the values for all attributes of an object of class **T**
 - Essentially, allocate a new object
 - Initialize those locations with the default values of attributes
 - Evaluate the initializers and set the resulting attribute values
 - Return the newly allocated object

Prof. Neula CS 164

26

Default Values

- For each class A there is a default value denoted by D_A
 - $D_{\text{int}} = \text{Int}(0)$
 - $D_{\text{bool}} = \text{Bool}(\text{false})$
 - $D_{\text{string}} = \text{String}(0, "")$
 - $D_A = \text{void}$ (for another class A)

More Notation

- For a class A we write $\text{class}(A) = (a_1 : T_1 \leftarrow e_1, \dots, a_n : T_n \leftarrow e_n)$ where
 - a_i are the attributes (including the inherited ones)
 - T_i are their declared types
 - e_i are the initializers

Operational Semantics of new

- Observation: `new SELF_TYPE` allocates an object with the same dynamic type as `self`

$$\begin{array}{l} T_0 = \text{if } T == \text{SELF_TYPE} \text{ and } \text{so} = X(\dots) \text{ then } X \text{ else } T \\ \text{class}(T_0) = (a_1 : T_1 \leftarrow e_1, \dots, a_n : T_n \leftarrow e_n) \\ l_i = \text{newloc}(S) \text{ for } i = 1, \dots, n \\ v = T_0(a_1 = l_1, \dots, a_n = l_n) \\ E' = [a_1 : l_1, \dots, a_n : l_n] \\ S_1 = S[D_{T_1}/l_1, \dots, D_{T_n}/l_n] \\ v, E', S_1 \vdash \{ a_1 \leftarrow e_1; \dots; a_n \leftarrow e_n; \} : v_n, S_2 \end{array}$$

$$\text{so}, E, S \vdash \text{new } T : v, S_2$$

Operational Semantics of new. Notes.

- The first three lines allocate the object
- The rest of the lines initialize it
 - By evaluating a sequence of assignments
- State in which the initializers are evaluated
 - Self is the current object
 - Only the attributes are in scope (same as in typing)
 - Starting value of attributes are the default ones
- The side-effect of initialization is preserved

Operational Semantics of Method Dispatch

- Consider the expression $e_0.f(e_1, \dots, e_n)$
- Informal semantics:
 - Evaluate the arguments in order e_1, \dots, e_n
 - Evaluate e_0 to the target object
 - Let X be the dynamic type of the target object
 - Fetch from X the definition of f (with n args.)
 - Create n new locations and an environment that maps f 's formal arguments to those locations
 - Initialize the locations with the actual arguments
 - Set $self$ to the target object and evaluate f 's body

Prof. Necula CS 164

31

More Notation

- For a class A and a method f of A (possibly inherited) we write:

$\text{impl}(A, f) = (x_1, \dots, x_n, e_{\text{body}})$ where

- x_i are the names of the formal arguments
- e_{body} is the body of the method

Prof. Necula CS 164

32

Operational Semantics of Dispatch

$$\begin{array}{l}
 SO, E, S \vdash e_1 : v_1, S_1 \\
 SO, E, S_1 \vdash e_2 : v_2, S_2 \\
 \dots \\
 SO, E, S_{n-1} \vdash e_n : v_n, S_n \\
 SO, E, S_n \vdash e_0 : v_0, S_{n+1} \\
 v_0 = X(a_1 = l_1, \dots, a_m = l_m) \\
 \text{impl}(X, f) = (x_1, \dots, x_n, e_{\text{body}}) \\
 l_{x_i} = \text{newloc}(S_{n+1}) \text{ for } i = 1, \dots, n \\
 E' = [x_1 : l_{x_1}, \dots, x_n : l_{x_n}, a_1 : l_1, \dots, a_m : l_m] \\
 S_{n+2} = S_{n+1}[v_1/l_{x_1}, \dots, v_n/l_{x_n}] \\
 v_0, E', S_{n+2} \vdash e_{\text{body}} : v, S_{n+3} \\
 \hline
 SO, E, S \vdash e_0.f(e_1, \dots, e_n) : v, S_{n+3}
 \end{array}$$

Operational Semantics of Dispatch. Notes.

- The body of the method is invoked with
 - E mapping formal arguments and self's attributes
 - S like the caller's except with actual arguments bound to the locations allocated for formals
- The notion of the activation frame is implicit
 - New locations are allocated for actual arguments
- The semantics of static dispatch is similar except the implementation of f is taken from the specified class

Runtime Errors

Operational rules do not cover all cases

Consider for example the rule for dispatch:

$$\frac{\begin{array}{l} \dots \\ s_0, E, S_n \vdash e_0 : v_0, S_{n+1} \\ v_0 = X(a_1 = l_1, \dots, a_m = l_m) \\ \text{impl}(X, f) = (x_1, \dots, x_n, e_{\text{body}}) \\ \dots \end{array}}{s_0, E, S \vdash e_0.f(e_1, \dots, e_n) : v, S_{n+3}}$$

What happens if $\text{impl}(X, f)$ is not defined?

Cannot happen in a well-typed program (Type safety theorem)

Prof. Necula CS 164

35

Runtime Errors (Cont.)

- There are some runtime errors that the type checker does not try to prevent
 - A dispatch on void
 - Division by zero
 - Substring out of range
 - Heap overflow
- In such case the execution must abort gracefully
 - With an error message not with segfault

Prof. Necula CS 164

36

Conclusions

- Operational rules are very precise
 - Nothing is left unspecified
- Operational rules contain a lot of details
 - Read them carefully
- Most languages do not have a well specified operational semantics
- When portability is important an operational semantics becomes essential
 - But not always using the notation we used for Cool