

## Operational Semantics of Cool

### Lecture 13

Prof. Aiken CS 143 Lecture 13

1

## Lecture Outline

---

- COOL operational semantics
- Motivation
- Notation
- The rules

Prof. Aiken CS 143 Lecture 13

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. Aiken CS 143 Lecture 13

3

## Evaluation Rules So Far

---

- We have specified evaluation rules indirectly
  - The compilation of Cool to a stack machine
  - The evaluation rules of the stack machine
- This is a complete description
  - Why isn't it good enough?

Prof. Aiken CS 143 Lecture 13

4

## Assembly Language Description of Semantics

---

- Assembly-language descriptions of language implementation have irrelevant detail
  - Whether to use a stack machine or not
  - Which way the stack grows
  - How integers are represented
  - The particular instruction set of the architecture
- We need a complete description
  - But not an overly restrictive specification

Prof. Aiken CS 143 Lecture 13

5

## Programming Language Semantics

---

- A multitude of ways to specify semantics
  - All equally powerful
  - Some more suitable to various tasks than others
- Operational semantics
  - Describes program evaluation via execution rules
    - on an abstract machine
  - Most useful for specifying implementations
  - This is what we use for Cool

Prof. Aiken CS 143 Lecture 13

6

## Other Kinds of Semantics

- Denotational semantics
  - Program's meaning is a mathematical function
  - Elegant, but introduces complications
    - Need to define a suitable space of functions
- Axiomatic semantics
  - Program behavior described via logical formulae
    - If execution begins in state satisfying  $X$ , then it ends in state satisfying  $Y$
    - $X, Y$  formulas
  - Foundation of many program verification systems

Prof. Aiken CS 143 Lecture 13

7

## Introduction to Operational Semantics

- Once again we introduce a formal notation
- Logical rules of inference, as in type checking

Prof. Aiken CS 143 Lecture 13

8

## Inference Rules

- 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*, expr.  $e$  evaluates to value  $v$ )

Prof. Aiken CS 143 Lecture 13

9

## Example Operational Semantics Rule

- Example:
$$\frac{\text{Context} \vdash e_1 : 5 \quad \text{Context} \vdash e_2 : 7}{\text{Context} \vdash e_1 + e_2 : 12}$$
- The result of evaluating an expression can depend on the result of evaluating its subexpressions
- The rules specify everything that is needed to evaluate an expression

Prof. Aiken CS 143 Lecture 13

10

## Contexts are Needed for 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 evaluation
- We track variables and their values with:
  - An *environment* : tells us *where* in memory a variable is stored
  - A *store* : tells us *what* is in memory

Prof. Aiken CS 143 Lecture 13

11

## 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:
$$E = [a : l_1, b : l_2]$$
- $E(a)$  looks up variable  $a$  in environment  $E$

Prof. Aiken CS 143 Lecture 13

12

## Stores

- A store maps memory locations to values
- Example:  
 $S = [l_1 \rightarrow 5, l_2 \rightarrow 7]$
- $S(l_i)$  is the contents of a location  $l_i$  in store  $S$
- $S' = S[12/l_1]$  defines a store  $S'$  such that  
 $S'(l_1) = 12$  and  $S'(l) = S(l)$  if  $l \neq l_1$

Prof. Aiken CS 143 Lecture 13

13

## Cool Values

- Cool values are objects
  - All objects are instances of some class
- $X(a_1 = l_1, \dots, a_n = l_n)$  is a Cool object where
  - $X$  is the class of the object
  - $a_i$  are the attributes (including inherited ones)
  - $l_i$  is the location where the value of  $a_i$  is stored

Prof. Aiken CS 143 Lecture 13

14

## Cool Values (Cont.)

- Special cases (classes without attributes)
  - $\text{Int}(5)$  the integer 5
  - $\text{Bool}(\text{true})$  the boolean true
  - $\text{String}(4, \text{"Cool"})$  the string "Cool" of length 4
- There is a special value **void** of type **Object**
  - No operations can be performed on it
  - Except for the test **isvoid**
  - Concrete implementations might use NULL here

Prof. Aiken CS 143 Lecture 13

15

## Operational Rules of Cool

- The evaluation judgment is  
 $so, E, S \vdash e : v, S'$
- read:
  - Given  $so$  the current value of **self**
  - And  $E$  the current variable environment
  - And  $S$  the current store
  - If the evaluation of  $e$  terminates then
  - The return value is  $v$
  - And the new store is  $S'$

Prof. Aiken CS 143 Lecture 13

16

## Notes

- "Result" of evaluation is a value and a store
  - New store models the side-effects
- Some things don't change
  - The variable environment
  - The value of **self**
  - The operational semantics allows for non-terminating evaluations

Prof. Aiken CS 143 Lecture 13

17

## Operational Semantics for Base Values

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

$\frac{i \text{ is an integer literal}}{so, E, S \vdash i : \text{Int}(i), S}$        $\frac{s \text{ is a string literal} \quad n \text{ is the length of } s}{so, E, S \vdash s : \text{String}(n, s), S}$

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

Prof. Aiken CS 143 Lecture 13

18

## Operational Semantics of Variable References

$$\frac{E(id) = l_{id} \quad S(l_{id}) = v}{so, E, S \vdash id : v, S}$$

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

Prof. Aiken CS 143 Lecture 13

19

## Operational Semantics for Self

- A special case:

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

Prof. Aiken CS 143 Lecture 13

20

## Operational Semantics of Assignment

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

- Three step process
  - Evaluate the right hand side
    - a value  $v$  and new store  $S_1$
  - Fetch the location of the assigned variable
  - The result is the value  $v$  and an updated store

Prof. Aiken CS 143 Lecture 13

21

## Operational Semantics of Conditionals

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

- 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 **Bool**. Why?

Prof. Aiken CS 143 Lecture 13

22

## Operational Semantics of Sequences

$$\frac{so, E, S \vdash e_1 : v_1, S_1 \quad so, E, S_1 \vdash e_2 : v_2, S_2 \quad \dots \quad so, E, S_{n-1} \vdash e_n : v_n, S_n}{so, E, S \vdash \{ e_1; \dots; e_n \} : v_n, S_n}$$

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

Prof. Aiken CS 143 Lecture 13

23

## Operational Semantics of while (I)

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

- If  $e_1$  evaluates to **false** the loop terminates
  - With the side-effects from the evaluation of  $e_1$
  - And with result value **void**
- Type checking ensures  $e_1$  evaluates to a **Bool**

Prof. Aiken CS 143 Lecture 13

24

## Operational Semantics of **while** (II)

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

- 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. Aiken CS 143 Lecture 13

25

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

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

- In what context should  $e_2$  be evaluated?
  - Environment like  $E$  but with a new binding of  $id$  to a fresh location  $l_{\text{new}}$
  - Store like  $S_1$  but with  $l_{\text{new}}$  mapped to  $v_1$

Prof. Aiken CS 143 Lecture 13

26

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

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

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

Prof. Aiken CS 143 Lecture 13

27

## Operational Semantics of **new**

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

Prof. Aiken CS 143 Lecture 13

28

## Default Values

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

Prof. Aiken CS 143 Lecture 13

29

## 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

Prof. Aiken CS 143 Lecture 13

30

## Operational Semantics of new

- **new SELF\_TYPE** allocates an object with the same dynamic type as **self**

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

Prof. Aiken CS 143 Lecture 13

31

## Notes on Operational Semantics of new.

- The first three steps allocate the object
- The remaining steps 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)
  - Initial values of attributes are the defaults

Prof. Aiken CS 143 Lecture 13

32

## Operational Semantics of Method Dispatch

- Informal semantics of  $e_0.f(e_1, \dots, e_n)$ 
  - 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. Aiken CS 143 Lecture 13

33

## 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
- $e_{\text{body}}$  is the body of the method

Prof. Aiken CS 143 Lecture 13

34

## Operational Semantics of Dispatch

$$\begin{aligned}
 so, E, S &\vdash e_1 : v_1, S_1 \\
 so, E, S_1 &\vdash e_2 : v_2, S_2 \\
 &\dots \\
 so, E, S_{n-1} &\vdash e_n : v_n, S_n \\
 so, E, S_n &\vdash e_0 : v_0, S_{n+1} \\
 v_0 &= X(a_1 = l_1, \dots, a_m = l_m) \\
 \text{impl}(X, f) &= (x_1, \dots, x_n, e_{\text{body}}) \\
 l_{x_i} &= \text{newloc}(S_{n+1}) \text{ for } i = 1, \dots, n \\
 E' &= [a_1 : l_1, \dots, a_m : l_m][X_1/l_{x_1}, \dots, X_n/l_{x_n}] \\
 S_{n+2} &= S_{n+1}[V_1/l_{x_1}, \dots, V_n/l_{x_n}] \\
 \hline
 v_0, E', S_{n+2} &\vdash e_{\text{body}} : v, S_{n+3} \\
 \hline
 so, E, S &\vdash e_0.f(e_1, \dots, e_n) : v, S_{n+3}
 \end{aligned}$$

Prof. Aiken CS 143 Lecture 13

35

## Notes on Operational Semantics of Dispatch

- 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 frame is implicit
  - New locations are allocated for actual arguments
- The semantics of static dispatch is similar

Prof. Aiken CS 143 Lecture 13

36

## Runtime Errors

Operational rules do not cover all cases

Consider the dispatch example:

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

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

**Cannot happen in a well-typed program**

Prof. Aiken CS 143 Lecture 13

37

## Runtime Errors (Cont.)

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

Prof. Aiken CS 143 Lecture 13

38

## Conclusions

- Operational rules are very precise & detailed
  - Nothing is left unspecified
  - Read them carefully
- Most languages do not have a well specified operational semantics
- When portability is important an operational semantics becomes essential

Prof. Aiken CS 143 Lecture 13

39