

## Operational Semantics of Cool

Adapted from Lectures by  
Prof. Alex Aiken and George Necula (UCB)

## Lecture Outline

- COOL operational semantics
  - Motivation
  - Notation
  - The rules

## 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

## Evaluation Rules So Far

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

## Assembly Language Description of Semantics

- Assembly-language descriptions of language implementation have too much "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

## 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

## 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 terminates with a sorted array
  - The foundation of many program verification systems

CS781(Prasad)

L24CG

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
$$\text{Context} \sqsupset e : C$$
(in the given **context**, expression  $e$  has type  $C$ )
- We try something similar for evaluation
$$\text{Context} \sqsupset e : v$$
(in the given **context**, expression  $e$  evaluates to value  $v$ )

CS781(Prasad)

L24CG

8

## Example of Inference Rule for Operational Semantics

- Example:
$$\frac{\text{Context} \sqsupset e_1 : 5 \quad \text{Context} \sqsupset e_2 : 7}{\text{Context} \sqsupset e_1 + e_2 : 12}$$
- In general, the result of evaluating an expression depends on the result of evaluating its sub-expressions
- The logical rules specify everything that is needed to evaluate an expression

CS781(Prasad)

L24CG

9

## What Contexts Are Needed?

- 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

CS781(Prasad)

L24CG

10

## 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  $E(a)$

CS781(Prasad)

L24CG

11

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

CS781(Prasad)

L24CG

12

## 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)
  - $\text{Int}(5)$  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  $\text{void}$  of type  $\text{Object}$ 
  - No operations can be performed on it
  - Except for the test  $\text{isvoid}$
  - Concrete implementations might use NULL here

## Operational Rules of Cool

- The evaluation judgment is
 
$$so, E, S \sqsupset e : v, S'$$
 read:
  - Given  $so$  the current value of  $\text{self}$
  - 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  $S'$

## Notes

- The "result" of evaluating an expression is a value and a new *store*
- The *store* changes 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

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

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

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

## Operational Semantics of Variable References

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

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

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

## Operational Semantics of Assignment

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

- A three step process
  - Evaluate the right hand side  
 $\Rightarrow$  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

CS781(Prasad)

L24CG

19

## Example

```
class Main {
  Int p <- 6;
  Int q <- p;
}
```

ENV:  $p \rightarrow 111000X \quad q \rightarrow 111004X$   
 STORE:  $111000X \rightarrow \text{Int}(6) \quad 111004X \rightarrow \text{Int}(6)$

- Copy semantics implicit in this value copying.

CS781(Prasad)

L24CG

20

## Example

```
class C { Int j <- 6; }
class Main { C p <- new C; C q <- p; }
```

- ENV:  $p \rightarrow 111000X$   
 $q \rightarrow 111004X$
- object mini-env :  $C:j \rightarrow 111008X$
- STORE:  $111000X \rightarrow C(j:111008X)$   
 $111004X \rightarrow C(j:111008X)$   
 $111008X \rightarrow \text{Int}(6)$
- Reference semantics; Stack vs Heap

CS781(Prasad)

L24CG

21

## Operational Semantics of Conditionals

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

- 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

CS781(Prasad)

L24CG

22

## Operational Semantics of Sequences

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

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

CS781(Prasad)

L24CG

23

## Operational Semantics of while (I)

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

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

CS781(Prasad)

L24CG

24

## Operational Semantics of while (II)

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

- 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

CS781(Prasad)

L24CG

25

## Operational Semantics of let Expressions (I)

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

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

CS781(Prasad)

L24CG

26

## Operational Semantics of let Expressions (II)

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

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

CS781(Prasad)

L24CG

27

## Operational Semantics of new

- Consider the expression  $\text{new } T$
- 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

CS781(Prasad)

L24CG

28

## Default Values

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

CS781(Prasad)

L24CG

29

## Operational Semantics of new

- **Observation:**  $\text{new SELF\_TYPE}$  allocates an object with the same dynamic type as  $\text{self}$

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

CS781(Prasad)

L24CG

30

## 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

## 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

## Operational Semantics of Dispatch

$$\begin{aligned}
 & s_0, E, S \sqsubseteq e_1 : v_1, S_1 \\
 & s_0, E, S_1 \sqsubseteq e_2 : v_2, S_2 \\
 & \dots \\
 & s_0, E, S_{n-1} \sqsubseteq e_n : v_n, S_n \\
 & s_0, E, S_n \sqsubseteq e_0 : v_0, S_{n+1} \\
 & v_0 = X(a_1 = l_{x_1}, \dots, a_m = l_{x_m}) \\
 & \text{impl}(X, f) = (x_1, \dots, x_n, e_{\text{body}}) \\
 & l_{x_i} = \text{newloc}(S_{n+1}) \text{ for } i = 1, \dots, n \\
 & E' = [x_1 : l_{x_1}, \dots, x_n : l_{x_n}, a_1 : l_{x_1}, \dots, a_m : l_{x_m}] \\
 & S_{n+2} = S_{n+1}[v_1/l_{x_1}, \dots, v_n/l_{x_n}] \\
 & \underline{v_0, E', S_{n+2} \sqsubseteq e_{\text{body}} : v, S_{n+3}} \\
 & s_0, E, S \sqsubseteq e_0.f(e_1, \dots, e_n) : v, S_{n+3}
 \end{aligned}$$

## 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

## Expression Evaluation Ordering

<pre> class A {   int f(A x) {1} }  class B extends A {   int f(A x) {2} }                     </pre>	<pre> class Main {   A a &lt;- new A;    a &lt;- f(a); // 1    a &lt;- f(a &lt;- new B); // 2 }                     </pre>
---	--

## Runtime Errors

---

Operational rules do not cover all cases

Consider the dispatch example:

```
...
so, E, Sn ⊢ e0 : v0, Sn+1
v0 = X(a1 = l1, ..., am = lm)
impl(X, f) = (x1, ..., xn, ebody)
...
so, E, S ⊢ e0.f(e1, ..., en) : v, Sn+3
```

What happens if `impl(X, f)` is not defined?

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

## 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 segmentation fault

## 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