Operational Semantics of Cool

Lecture 13

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 ⇒ lexical analysis
  - The grammar ⇒ syntactic analysis
  - The typing rules ⇒ semantic analysis
  - The evaluation rules
    ⇒ code generation and optimization

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?

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

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

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

Introduction to Operational Semantics

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

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)

Example Operational Semantics Rule

- Example:

$$\frac{\text{Context} \vdash e_1 : 5 \qquad \text{Context} \vdash e_2 : 7}{\text{Context} \vdash e_1 + e_2 : 12}$$

- 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

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

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

Stores

- A store maps memory locations to values
- Example:

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

- $S(l_1)$ is the contents of a location l_1 in store S

- $S' = S[12/l_1]$ defines a store S' such that

  $S'(l_1) = 12$ and $S'(l) = S(l)$ if $l \neq l_1$

Cool Values

- Cool values are objects
  - All objects are instances of some class

- $X(a_1 = l_1, \ldots, a_n = l_n)$ 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

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 of type Object
  - 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 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

- "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

Operational Semantics for Base Values

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

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

- No side effects in these cases

  (the store does not change)

Operational Semantics of Variable References

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

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

Operational Semantics for Self

- A special case:

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

Operational Semantics of Assignment

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

- 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

Operational Semantics of Conditionals

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

- 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?

Operational Semantics of Sequences

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

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

Operational Semantics of while (I)

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

- 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

Operational Semantics of while (II)

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

- 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

Operational Semantics of let Expressions (I)

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

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

Operational Semantics of let Expressions (II)

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

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

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

Default Values

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

More Notation

- For a class A we write
  $class(A) = (a_1 : T_1 \leftarrow e_1, \ldots, a_n : T_n \leftarrow e_n)$ 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

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

$$
\frac{
\begin{array}{l}
T_0 = \text{if } (T == \text{SELF\_TYPE and } so = X(\ldots)) \text{ then } X \text{ else } T \\
class(T_0) = (a_1 : T_1 \leftarrow e_1, \ldots, a_n : T_n \leftarrow e_n) \\
l_i = newloc(S) \text{ for } i = 1, \ldots, n \\
v = T_0(a_1 = l_1, \ldots, a_n = l_n) \\
S_1 = S[D_{T1}/l_1, \ldots, D_{Tn}/l_n] \\
E' = [a_1 : l_1, \ldots, a_n : l_n] \\
v, E', S_1 \vdash \{ a_1 \leftarrow e_1; \ldots; a_n \leftarrow e_n; \} : v_n, S_2
\end{array}
}{so, E, S \vdash \text{new } T : v, S_2}
$$

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

Operational Semantics of Method Dispatch

- Informal semantics of $e_0.f(e_1,\ldots,e_n)$
  - Evaluate the arguments in order $e_1,\ldots,e_n$
  - 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:

$impl(A, f) = (x_1, \ldots, x_n, e_{body})$ where
  - x_i are the names of the formal arguments
  - e_{body} is the body of the method

Operational Semantics of Dispatch

$$
\frac{
\begin{array}{l}
so, E, S \vdash e_1 : v_1, S_1 \\
so, E, S_1 \vdash e_2 : v_2, S_2 \\
\ldots \\
so, E, S_{n-1} \vdash e_n : v_n, S_n \\
so, E, S_n \vdash e_0 : v_0, S_{n+1} \\
v_0 = X(a_1 = l_1, \ldots, a_m = l_m) \\
impl(X, f) = (x_1, \ldots, x_n, e_{body}) \\
l_{xi} = newloc(S_{n+1}) \text{ for } i = 1, \ldots, n \\
E' = [a_1 : l_1, \ldots, a_m : l_m][x_1/l_{x1}, \ldots, x_n/l_{xn}] \\
S_{n+2} = S_{n+1}[v_1/l_{x1}, \ldots, v_n/l_{xn}] \\
v_0, E', S_{n+2} \vdash e_{body} : v, S_{n+3}
\end{array}
}{so, E, S \vdash e_0.f(e_1, \ldots, e_n) : v, S_{n+3}}
$$

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

Runtime Errors

Operational rules do not cover all cases

Consider the dispatch example:

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

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

Cannot happen in a well-typed program

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

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