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
  - Somehow give rules to specify what kind of computation a particular expression does
- The definition of a programming language:
  - For lexical analysis ⇒ tokens
  - For syntactic analysis ⇒ grammar
  - For semantic analysis ⇒ formal type rules
  - For code generation and optimization
    ⇒ evaluation rules (these guide code gen and opt.)

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
  - You could take the generated assembly code, run it on the machine, and see what the program does. This would be a valid description of the behavior of the program
  - Why isn't it good enough?
    - I.e., why isn't it good enough to just have a code generator describe what is supposed to happen?

Motivation

- This may be difficult to appreciate without having written a few compilers.
- In a nutshell, people, through hard experience, have learned that…

Assembly Language Description of Semantics

- Assembly-language descriptions of language implementations have a lot of irrelevant detail -- there is a lot of unnecessary stuff you have to say when using such a complete executable description of a program
  - Whether to use a stack machine or not
  - Which way the stack grows
  - How integers are represented
  - The particular instruction set of the architecture
- None of these are intrinsic to any particular programming language

Assembly Language Description of Semantics

- Assembly-language descriptions of language implementations have a lot of irrelevant detail -- there is a lot of unnecessary stuff you have to say when using such a complete executable description of a program
  - Whether to use a stack machine or not
  - Which way the stack grows
  - How integers are represented
  - The particular instruction set of the architecture
- Moreover, these are ONE way to describe the language, but we don't want it to be the only way

Assembly Language Description of Semantics

- We need a complete description
  - But not an overly restrictive specification
    - I.e., we want a description that allows a variety of implementations
- When people have not done this (tried to find a relatively high-level way to describe the behavior of the language) they have invariably ended up having to run the program on a reference implementation

So What?

- Reference implementation not completely correct themselves
  - They have bugs
  - There are artifacts of the particular way in which it was implemented.
    - These artifacts, because there is no better way of defining some behavior, become an unintended part of the language! A part that you may not want!
  - You don't really want aspects of your language to be defined based on accidents that occurred because of the way the language was implemented for the first time

Programming Language Semantics

- Many ways to specify semantics
  - All equally powerful
  - Some more suitable to various tasks than others

- We'll use *operational semantics*
  - Describes program evaluation via execution rules on an abstract machine
    - Think of a very high-level kind of code generation
  - 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
    - Program text is mapped to a mathematical function that goes from inputs to outputs
  - Elegant, but introduces complications we don't really need to consider for purposes of defining an implementation
    - Need to define a suitable space of functions

Other Kinds of Semantics

- *Axiomatic semantics*
  - Program behavior described via logical formulae
    - If execution begins in state satisfying X, then it ends in state satisfying Y
    - X and Y are formulas in some logic
  - Foundation of many program verification systems

Introduction to Operational Semantics

- Once again we introduce a formal notation
- Logical rules of inference (as we used in type checking)
  - With some twists

Inference Rules

- Recall the typing judgment

  $$\text{Context} \vdash e : C$$

  In the given context, expression e has type C

- We use something similar for evaluation

  $$\text{Context} \vdash e : v$$

  In the given context, expr. e evaluates to value v

  (Context is different: evaluation context as opposed to type context)

Example Operational Semantics Rule

- Example:

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

- Suppose that within the given Context (same for all three expressions) and using our rules (which I have not yet disclosed), we could show the hypotheses to be true. Then certainly the conclusion would be true.

Example Operational Semantics Rule

- Example:

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

- What is the context giving?
  - Well, consider what it did with type checking: *gave information about the free variables*
  - Here, it needs to *give information about the values of the free variables* that appear in the subexpressions

Example Operational Semantics Rule

- Example:

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

- 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

Note this use of term is not same as in type checking

- We track variables and their values with:
  - An *environment* : tells us *where* in memory a variable is stored
    - Technically a mapping from variables to memory locations
  - A *store* : tells us *what* is in memory
    - Technically a mapping from memory locations to values

Variable Environments (and Related Notation)

- A variable environment maps variable names to locations
  - Keeps track of which variables are in scope
  - Tells in where those variables are
  - It is a list of variable:location pairs
- Example:

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

- $E(a)$ looks up variable a in environment E
  - Here, variable a is at location l_1
  - Here, variable b is at location l_2

Variable Environments (and Related Notation)

- A variable environment maps variable names to locations
  - Keeps track of which variables are in scope
  - Tells in where those variables are
  - It is a list of variable:location pairs
- Example:

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

- E(a) looks up variable a in environment E
  - E keeps track of the variables that are in scope, so the only variables that E mentions are those that are in scope in the expressions being evaluated

Stores

arrow is used so that stores look a little different from environments. helps prevent confusing the two

- A store maps memory locations to values
- Example:

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

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

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

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

the replace or update operation

Cool Values

- Cool values are objects
  - Which are, of course, generally a bit more complicated than integers
  - 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
- This is a complete description of the object, since once we know where the variables are located, we can use store to look up values

Cool Values (Cont.)

- Special cases (classes without attributes) and special ways of writing them

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
    - Can't dispatch to void – gives a run-time error
  - 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'
    - Since e might have assignments in it that update the memory

So always read "if e terminates…"

Notes

- "Result" of evaluation is a value and a new store
  - New store models the side-effects
- Some things don't change
  - E - The variable environment
  - so - The current self object
    - These make sense: we can't update the self object in COOL, nor do we have access, in any form, to relocations of variables stored. So these are invariant under evaluation
  - The operational semantics allows for non-terminating evaluations
    - but judgement only holds if the evaluation of e terminates

Notes

- "Result" of evaluation is a value and a new store
  - New store models the side-effects
- Some things don't change
  - E - The variable environment
  - so - The current self object
    - Note: attributes of self object might change! It is location and layout of attributes that do not change
  - The operational semantics allows for non-terminating evaluations
    - but judgement only holds if the evaluation of e termnates

Operational Semantics for Base Values

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

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

$$\frac{\text{i is an integer literal}}{so, E, S \vdash i : Int(i), S}$$

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

- 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

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

note two parts: identifier being evaluated and an expression that gives the new value

- 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 Addition

$$\frac{\begin{array}{c} so, E, S \vdash e_1 : v_1, S_1 \\ so, E, S_1 \vdash e_2 : v_2, S_2 \end{array}}{so, E, S \vdash e_1 + e_2 : v_1 + v_2, S_2}$$

- Note the stores tell the order in which you have to evaluate the expressions:
  - Because e_1 is evaluated in the same store as the overall expression, e_1 must be evaluated first
  - Because e_2 is evaluated in the store produced by evaluating e_1, e_2 must be evaluated after e_1
  - Finally, because the overall value ends with store S_2, e_2 must be the last thing evaluated

Operational Semantics of Conditionals

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

- 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

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

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

Example

- Consider the expression $\{ X \leftarrow 7 + 5; 4;\}$

$$\frac{}{so, [x: I], [l \leftarrow 0] \vdash \{ x \leftarrow 7 + 5; 4; \}}$$

Example

- Consider the expression $\{ X \leftarrow 7 + 5;\ 4;\}$

$$\frac{so, [x:I], [I \leftarrow 0] \vdash x : 7 + 5 \qquad so, [x:I], [\ ?\ ] \vdash 4}{so, [x: I], [I \leftarrow 0] \vdash \{ x \leftarrow 7 + 5; 4; \}}$$

Example

- Consider the expression { X ← 7 + 5; 4;}

$$
\dfrac{
\dfrac{
\begin{array}{l}
\dfrac{
\begin{array}{l}
so, [x{:}l], [l \leftarrow 0] \vdash 7 : Int(7), [l \leftarrow 0] \\
so, [x{:}l], [l \leftarrow 0] \vdash 5 : Int(5), [l \leftarrow 0]
\end{array}
}{
so, [x{:}l], [l \leftarrow 0] \vdash 7 + 5 : Int(12), [l \leftarrow 0]
} \\
[l \leftarrow 0](12/l) = [l \leftarrow 12]
\end{array}
}{
so, [x{:}l], [l \leftarrow 0] \vdash x \leftarrow 7 + 5 : Int(12), [l \leftarrow 12]
}
\quad
so, [x{:}l], [l \leftarrow 12] \vdash 4 : Int(4), [l \leftarrow 12]
}{
so, [x{:}\ l], [l \leftarrow 0] \vdash \{\ x \leftarrow 7 + 5;\ 4;\ \} : Int(4), [l \leftarrow 12]
}
$$

Operational Semantics of while (I)

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

Note the resulting store is whatever resulted from evaluating the predicate

- 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 ($S \to S_1 \to S_2 \to S_3$)
- 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:

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

So far

- Some complicated stuff, but not the two most complex operations:
  - Allocation of a new object
  - Dynamic dispatch
  - So, onward…

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
  - Set attributes with their default values
    - We'll see in a minute what these attributes are, and why we need to set the attributes to defaults
  - Evaluate the initializers and set the resulting attribute values
  - Return the newly allocated object

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
  - Set attributes with their default values
    - We'll see in a minute what these attributes are, and why we need to set the attributes to defaults
  - Evaluate the initializers and set the resulting attribute values
  - Return the newly allocated object

Note: quite a bit more than just allocating a little bit of memory
Actually much computation occurring

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

  $\text{class}(A) = (a_1 : T_1 \leftarrow e_1, \ldots, a_n : T_n \leftarrow e_n)$ where

  - a_i are the attributes (including the inherited ones)
    - attributes listed in "greatest ancestor first" order
    - I.e., if $C \leq B \leq A$, then in call class(C), attributes of A listed first, then attributes of B, then attributes of C
    - For given class, attributes listed in order they appear in text
  - T_i are their declared types
  - e_i are the initializers
  - Note that class is a function. It takes a class name and returns the list of attributes of that class

Operational Semantics of new

- new SELF_TYPE allocates an object with the same dynamic type as self (this type is denoted here by X)

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

Note E' has no relation to E

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
    - Need the defaults because the attributes are in scope inside their own initializers (might need to read an attribute in order to finish computing its initial value)

Notes on Operational Semantics of **new**.

- Note that it is not just COOL that has complicated semantics for the initialization of new objects…

- Every OO language has fairly complex semantics for the initialization of new objects

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:

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

- x_i are the names of the formal arguments
- e_{body} is the body of the method
- As with class, impl is a function

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