Operational Semantics of Cool

ICOM 4029
Lecture 10
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
  - And 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 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
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 sorts an array
  - The foundation of many program verification systems
Introduction to Operational Semantics

- Once, again we introduce a formal notation
  - Using logical rules of inference, just like for typing

- Recall the typing judgment
  \[ \text{Context ` } e : C \]
  (in the given context, expression \( e \) has type \( C \))

- We try something similar for evaluation
  \[ \text{Context ` } e : v \]
  (in the given context, expression \( e \) evaluates to value \( v \))
Example of Inference Rule for Operational Semantics

• Example:

\[
\begin{align*}
\text{Context } e_1 &: 5 \\
\text{Context } e_2 &: 7 \\
\hline \\
\text{Context } e_1 + e_2 &: 12
\end{align*}
\]

• 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 \( 1_2 \) to location \( l_1 \) we write \( S[1_2 / l_1] \)
  - This denotes a store \( S' \) such that
    \[ S'(l_1) = 1_2 \text{ and } S'(l) = S(l) \text{ if } l \neq l_1 \]
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 $X(a_1 = l_1, \ldots, a_n = l_n)$ 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
  \[ \text{so, } E, S \ ` e : v, S' \]

read:
- Given \text{so} the current value of the self object
- And \text{E} the current variable environment
- And \text{S} the current store
- If the evaluation of \text{e} terminates then
  - The returned value is \text{v}
  - And the new store is \text{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 `true : Bool(true), S

so, E, S `false : Bool(false), S

so, E, S `i : Int(i), S

s is a string literal
n is the length of s

so, E, S `s : String(n,s), S

• No side effects in these cases
  (the store does not change)
Operational Semantics of Variable References

\[
E(id) = l_{id} \\
S(l_{id}) = v \\
\text{so, } E, S \ ` id : v, S
\]

• Note the double lookup of variables
  - First from name to location
  - Then from location to value

• The store does not change

• A special case:

\[
\text{so, } E, S \ ` \text{ self : so, } S
\]
Operational Semantics of Assignment

so, $E, S \leftarrow e : v, S_1$

$E(id) = l_{id}$

$S_2 = S_1[v/l_{id}]$

so, $E, S \leftarrow id \leftarrow e : v, S_2$

- 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
Operational Semantics of Conditionals

so, E, S ` e₁ : Bool(true), S₁
so, E, S₁ ` e₂ : v, S₂
so, E, S ` if e₁ then e₂ else e₃ : v, S₂

• The “threading” of the store enforces an evaluation sequence
  - e₁ must be evaluated first to produce S₁
  - Then e₂ can be evaluated
• The result of evaluating e₁ is a boolean object
  - The typing rules ensure this
  - There is another, similar, rule for Bool(false)
Operational Semantics of Sequences

\[
\begin{align*}
\text{so, } & E, S \setminus e_1 : v_1, S_1 \\
\text{so, } & E, S_1 \setminus e_2 : v_2, S_2 \\
\text{…} & \\
\text{so, } & E, S_{n-1} \setminus e_n : v_n, S_n \\
\text{so, } & E, S \setminus \{ e_1; \ldots; e_n; \} : v_n, S_n
\end{align*}
\]

- Again the threading of the store expresses the intended evaluation sequence
- Only the last value is used
- But all the side-effects are collected
Operational Semantics of \textbf{while (I)}

\begin{align*}
\text{so, } E, S \ ` e_1 : \text{Bool(false)}, S_1 \\
\text{so, } E, S \ ` \text{while } e_1 \text{ loop } e_2 \text{ pool : void, } S_1
\end{align*}

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

- 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)

- 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 $l_{\text{new}}$
  - Store like $S_1$ but with $l_{\text{new}}$ mapped to $v_1$

\[
\begin{align*}
\text{so, } E, S \ ` e_1 : v_1, S_1 \\
\text{so, } ?, ?, ? ` e_2 : v, S_2 \\
\text{so, } E, S ` \text{let id : T } \leftarrow e_1 \text{ in } e_2 : v_2, S_2
\end{align*}
\]
Operational Semantics of \texttt{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 \texttt{newloc} as the dynamic memory allocation function
- The operational rule for \texttt{let}:

\[
\begin{align*}
&\text{so, } E, S \setminus e_1 : v_1, S_1 \\
&l_{\text{new}} = \text{newloc}(S_1) \\
&\text{so, } E[l_{\text{new}}/\text{id}], S_1[v_1/l_{\text{new}}] \setminus e_2 : v_2, S_2 \\
&\text{so, } E, S \setminus \text{let id : } T \leftarrow e_1 \text{ in } e_2 : v_2, S_2
\end{align*}
\]
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
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 another class $A$)
More Notation

• For a class $A$ we write

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

- $a_i$ 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

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

so, E, S \ `\ new T : v, S_2

- 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 $e_0.f(e_1,\ldots,e_n)$

• Informal semantics:
  - 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:

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

where

- $x_i$ are the names of the formal arguments
- $e_{\text{body}}$ is the body of the method
Operational Semantics of Dispatch

so, E, S ` e_1 : v_1, S_1
so, E, S_1 ` e_2 : v_2, S_2

...  
so, E, S_{n-1} ` e_n : v_n, S_n
so, E, S_n ` e_0 : v_0, S_{n+1}

v_0 = X(a_1 = l_1, ... , a_m = l_m)
impl(X, f) = (x_1, ..., x_n, e_{body})
l_{x_i} = newloc(S_{n+1}) for i = 1, ..., n
E' = [x_1 : l_{x_1}, ..., x_n : l_{x_n}, a_1 : l_1, ..., a_m : l_m]
S_{n+2} = S_{n+1}[v_1/l_{x_1}, ..., v_n/l_{x_n}]

v_0 , E' , S_{n+2} ` e_{body} : v, S_{n+3}

so, E, S ` e_0.f(e_1, ..., e_n) : v, S_{n+3}
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:

\[ \text{impl}(X, f) = (x_1, \ldots, x_n, e_{\text{body}}) \]

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

Cannot happen in a well-typed program \( \text{(Type safety theorem)} \)
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 segfault
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