Status

• Project 3 due on Friday
• Project 4 assigned on Friday. Due June 5, 11:55pm
• Office Hours this week are Wed at 4pm and Fri at 11am
• No Class (Memorial Day) on Monday, May 25
• Read Sec. 8.1 through 8.8 by Wednesday, May 27
• Final Exam Review 9a-11a on June 5 in Olson 105
Motivation

- We must specify for every Cool expression what happens when 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 many irrelevant details
- Whether to use a stack machine or not
- Which way the stack grows
- How integers are represented on a particular machine
- We need a complete specification that is architecture and OS independent and doesn’t overly restrict implementation choices.
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 (sec 13 of Cool manual)
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
Intro to Operational Semantics

- Once again we introduce a formal notation
  - Using logical rules of inference, just like for typing
- Recall the typing judgment
  \[ \text{Context} \vdash e : C \]
  (in the given context, expression e has type C)
- We try something similar for evaluation
  \[ \text{Context} \vdash e : v \]
  (in the given context, expression e evaluates to value v)
Example of Inference Rule for Operational Semantics

- Example:

  Context ⊢ e₁ : 5
  Context ⊢ e₂ : 7
  Context ⊢ e₁ + e₂ : 12

- 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 12 to location \( l_1 \) we write \( S[12/l_1] \)
- This denotes a new store \( S' \) such that for some location \( l \)
  \[ S'(l_1) = 12 \quad \text{and} \quad S'(l) = S(l) \text{ if } l \neq l_1 \]
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

- 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` (e.g., `isvoid expr`)
  - Concrete implementations might use NULL here
Operational Rules of Cool

• The evaluation judgment is
  \( \text{so, E, S} \vdash e : v, S' \)

• read:
  • Given the object “so” with the current value of \textit{self}
  • And \( E \) the current variable environment
  • And \( S \) the current store
  • If the evaluation of \( e \) terminates then
  • The returned 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

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

\[
\begin{align*}
\text{so, } E, S \vdash \text{true : } \text{Bool(true)}, S \\
\text{so, } E, S \vdash \text{false : } \text{Bool(false)}, S \\
\text{i is an integer literal} \\
\text{so, } E, S \vdash \text{i : Int(i)}, S \\
\text{s is a string literal} \\
\text{n is the length of s} \\
\text{so, } E, S \vdash \text{s : String(n,s)}, S
\end{align*}
\]
Operational Semantics of Variable References

\[ E(id) = l_id \]
\[ S(l_id) = v \quad \text{so, E, so, E, S} \]
\[ \text{so, E, S} \vdash \text{id : v, S} \]

• Note the double lookup of variables
  • First from name to location
  • Then from location to value
• The store does not change
• A special case:
  \[ \text{so, E, S} \vdash \text{self : so, S} \]
Operational Semantics of Assignment

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

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

• A three step process
  • Evaluate the right hand side
    ⇒ 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 \vdash e_1: \text{Bool}(true), S_1

\[
\text{so, E, S} \vdash e_2: v, S_2
\]

\[
\text{so, E, S} \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3: v, S_2
\]

- 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 boolean object
  - The typing rules ensure this
Operational Semantics of Sequences

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

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

so, E, S ⊢ e₁ : Bool(false), S₁

so, E, S ⊢ while e₁ loop e₂ pool : void, S₁

• If e₁ evaluates to Bool(false) then the loop terminates immediately
  • With the side-effects from the evaluation of e₁
  • And with result value void
• The typing rules ensure that e₁ evaluates to a boolean object
Operational Semantics of while(II)

\[ \text{so, } E, S \vdash e_1 : \text{Bool(true)}, S_1 \]
\[ \text{so, } E, S_1 \vdash e_2 : v, S_2 \]
\[ \text{so, } E, S_2 \vdash \text{while } e_1 \text{ loop } e_2 \text{ pool : void}, S_3 \]
\[ \text{so, } E, S \vdash \text{while } e_1 \text{ loop } e_2 \text{ pool : void}, S_3 \]

- 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 (updated store) is preserved
Operational Semantics of let Expressions (I)

\[
\begin{align*}
\text{so, } E, S \vdash e_1 : v_1, S_1 \\
\text{so, } ?, ? \vdash e_2 : v_2, S_2 \\
\hline
\text{so, } E, S \vdash \text{let id : } T \leftarrow e_1 \text{ in } e_2 : v_2, S_2
\end{align*}
\]

- 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 \( v_1 \)
Operational Semantics of let Expressions (II)

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

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

• Consider the expression $new\ T$

• Informal semantics (we’ll do formal ones in a moment)
  • Essentially, allocate a new object
  • Allocate new locations to hold the values for all attributes of an object of class $T$
  • 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 $D_A$
  • $D_{int} = Int(0)$
  • $D_{bool} = Bool(false)$
  • $D_{string} = String(0, "")$
  • $D_A = void$ (for another class A)
More Notation

• For a class $A$ we write the following mapping:

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

where

• $a_i$ are the attributes (including the inherited ones)
• $T_i$ are their declared types
• $e_i$ are the initializers
Operational Semantics of new

Obs.: new SELF_TYPE allocates an object with the same dynamic type as self

if T == SELF_TYPE and so = X(...) then T₀ = X else T₀ = T

class(T₀) = (a₁:T₁ ← e₁, ..., aₙ:Tₙ ← eₙ)

lᵢ = newloc(S) foreach i = 1,...,n

v = T₀(a₁ = l₁,...,aₙ = lₙ)

E' = [a₁: l₁, ..., aₙ: lₙ]

S₁ = S[Dₜ₁/l₁,...,Dₜₙ/lₙ]

v, E', S₁ ⊢ { a₁ ← e₁; ...; aₙ ← eₙ; } : vₙ, S₂

so, E, S ⊢ new T : v, S₂

- 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 the following mapping:

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

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 ⊢ e₁ : v₁, S₁
so, E, S₁ ⊢ e₂ : v₂, S₂
...
so, E, Sₙ₋₁ ⊢ eₙ : vₙ, Sₙ
so, E, Sₙ ⊢ e₀ : v₀, Sₙ₊₁
v₀ = X(a₁ = l₁, ..., aₘ = lₘ)
implementation(X, f) = (x₁, ..., xₙ, e_body)
lₓᵢ = newloc(Sₙ₊₁) for i = 1, ..., n
E’ = [x₁: lₓ₁, ..., xₙ: lₓₙ, a₁: l₁, ..., aₘ: lₘ]
Sₙ₊₂ = Sₙ₊₁[v₁/lₓ₁, ..., vₙ/lₓₙ]
v₀, E’, Sₙ₊₂ ⊢ e_body : v, Sₙ₊₃

so, E, S ⊢ e₀.f(e₁, ..., eₙ) : v, Sₙ₊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:

  \[ \ldots \]
  
  so, \( E, S_n \vdash e_0 : v_0, S_{n+1} \)
  
  \( v_0 = X(a_1 = l_1, \ldots, a_m = l_m) \)
  
  \( \text{implementation}(X, f) = (x_1, \ldots, x_n, e_{\text{body}}) \)
  
  \[ \ldots \]

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

- What happens if the mapping function \( \text{implementation}(X, f) \) is not defined?

- Cannot happen in a well-typed program (type safety theorem)
• 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 should abort gracefully
  • With an error message not with segfault
Conclusions

- Operational rules are very precise
- Nothing (that can be specified) is left unspecified
- Operational rules contain a lot of details
- Read them carefully
- Most languages do not have a well specified operational semantics (but Cool does)
- When portability is important an operational semantics becomes essential (hence portability problems)
- But not always using the notation we used for Cool
Finish Project 3!