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 ⇒ lexical analysis
  - The grammar ⇒ syntactic analysis
  - The typing rules ⇒ semantic analysis
  - The evaluation rules ⇒ 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

Introduction to Operational Semantics

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

\[
\text{Context} \vdash e : C
\]

(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:

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

- 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

What Contexts Are Needed?

- Contexts are needed to handle variables
- Consider the evaluation of \( y ← x + 1 \)
  - 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 store \( S' \) such that \( S'(l_1) = 12 \) and \( S'(l) = S(l) \) 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 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 $E, S \vdash e : v, S'$
- Given $E$ 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

- No side effects in these cases (the store does not change)
  - So, $E, S \vdash true : Bool(true), S$
  - So, $E, S \vdash false : Bool(false), S$
  - $i$ is an integer literal
    - So, $E, S \vdash i : Int(i), S$
  - $s$ is a string literal
    - So, $E, S \vdash s : String(n,s), S$

Operational Semantics of Variable References

- 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, $E, S \vdash self : so, S$
Operational Semantics of Assignment

- 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 \( S_2 = S_1[v/lid] \)
- The environment does not change

Example

```java
class Main {
    int p <- 6;
    int q <- p;
}
```

**ENV:**

- \( p \rightarrow 111000X \)
- \( q \rightarrow 111004X \)

**STORE:**

- \( 111000X \rightarrow \) int(6)
- \( 111004X \rightarrow \) int(6)

Operational Semantics of Conditionals

- 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

Example

```java
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:**

- \( 111000X \rightarrow C(j:111008X) \)
- \( 111004X \rightarrow \) int(6)
- Reference semantics; Stack vs Heap

Operational Semantics of Sequences

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

Example

```java
if e1 then e2 else e3 : v, S2
```

Operational Semantics of while (I)

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

Example

```java
while e1 loop e2 pool : void, S1
```

so, E, S \[ i : \text{Bool}(true), S_1 \]

so, E, S \[ i : \text{void}, S_2 \]

so, E, S \[ i : \text{void}, S_3 \]

so, E, S \[ i : \text{void}, S_n \]

so, E, S \[ \{ e_1; \ldots; e_n \} : \text{void}, S_n \]

so, E, S \[ \text{while } e_1 \text{ loop } e_2 \text{ pool : void}, S_1 \]
Operational Semantics of \texttt{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

\[
\begin{align*}
\text{so}, & \quad E, S \square e_1 : \text{Bool}(\text{true}), S_1 \\
\text{so}, & \quad E, S, S_1 \square e_2 : v, S_2 \\
\text{so}, & \quad E, S, S_2 \square \text{while } e_1 \text{ loop } e_2 \text{ pool} : \text{void}, S_3 \\
\end{align*}
\]

Operational Semantics of \texttt{let} Expressions (I)

- 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 $v_1$

\[
\begin{align*}
\text{so}, & \quad E, S \square e_1 : v_1, S_1 \\
\text{so}, & \quad ?, ?, ? \square e_2 : v, S_2 \\
\end{align*}
\]

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}:

\[
\begin{align*}
\text{so}, & \quad E, S \square e_1 : v_1, S_1 \\
\text{l}_{\text{new}} = & \text{newloc}(S_1) \\
\text{so}, & \quad E[l_{\text{new}}/\text{id}] , S_1[v_1/l_{\text{new}}] \\
\text{so}, & \quad E, S \square \text{let } \text{id} : T \leftarrow e_1 \text{ in } e_2 : v_2, S_2 \\
\end{align*}
\]

Operational Semantics of \texttt{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

\[
\begin{align*}
\text{so}, & \quad E, S \square e_1 : v_1, S_1 \\
\text{so}, & \quad ?, ?, ? \square e_2 : v, S_2 \\
\end{align*}
\]

Default Values

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

\[
\begin{align*}
\text{class}(A) = (a_1 : T_1 \leftarrow e_1, \ldots, a_n : T_n \leftarrow e_n) \quad \text{where}
\end{align*}
\]

- $a_i$ are the attributes (including the inherited ones)
- $T_i$ are their declared types
- $e_i$ are the initializers

Operational Semantics of \texttt{new}

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

\[
\begin{align*}
T_o & = \text{if } T == \text{SELF\_TYPE} \text{ and } \text{so} = X(\ldots) \text{ then } X \text{ else } T \\
class(T_o) & = (a_1 : T_1 \leftarrow e_1, \ldots, a_n : T_n \leftarrow e_n) \\
l_i & = \text{newloc}(S) \text{ for } i = 1, \ldots, n \\
v & = T_o(a_1 = l_1, \ldots, a_n = l_n) \\
e' & = E[l_1, \ldots, a_n] \\
S_1 & = S[D_{T_1}/l_1, \ldots, D_{T_n}/l_n] \\
v, e', S_1 & \square (a_1 \leftarrow e_1, \ldots, a_n \leftarrow e_n) : v, S_2 \\
\text{so}, & \quad E, S \square \text{new } T : v, S_2 \\
\end{align*}
\]

- 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 $e_0.f(e_1,...,e_n)$
- Informal semantics:
  - Evaluate the arguments in order $e_1,...,e_n$
  - 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:
  $$\text{impl}(A, f) = (x_1, \ldots, x_n, e_{\text{body}})$$
  where
  - $x_i$ are the names of the formal arguments
  - $e_{\text{body}}$ is the body of the method

Operational Semantics of Dispatch

- 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

```java
class A {
    int f(A x) {1}
}
class Main {
    A a <- new A;
    a <- f (a); // 1
    a <- f (a<-new B); //2
}
```
Runtime Errors

Operational rules do not cover all cases
Consider the dispatch example:

\[
\begin{align*}
\text{so}, & \ E, S_n \vdash e_0 : v_0, S_{n+1} \\
v_0 & = X(a_1 = 1_1, \ldots, a_m = 1_m) \\
\text{impl}(X, f) & = (x_1, \ldots, x_n, e_{\text{body}}) \\
\text{so}, & \ E, S \vdash e_0.f(e_1, \ldots, e_n) : v, S_{n+2}
\end{align*}
\]

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

\textit{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
So operational semantics can be pretty cool, but ultimately they're little more than specifying how to evaluate a program. In some ways, this isn't very satisfying: it focuses on "how" rather than "what" where we'd really like to be more declarative if possible. Denotational Semantics. Denotational semantics are how we can be "more declarative". Unlike operational and denoational semantics, I've never really run into axiomatic semantics outside my PL theory course. So you're better off looking it up on Wikipedia. Anyhow: the point is that we can define semantics for programming languages, and in fact there are a whole bunch of different ways to do it. 13. 5. Natural Operational Semantics Natural operational semantics (large step operational semantics) gives, for each program, the effect of the program $\alpha\epsilon$ usually the poststate as a function of the prestate and the effects of other commands. Notations: $S$: a state $s$, $s \rightarrow s'$: The program $S$, when run on input $s$, terminates in the state $s'$. Operational Semantics. Page 8. The operational semantics of while: Simple statements: $\epsilon$ Skip This repository contains the static semantics of cool programming language developed as part of the programming assignment of Principle of Compiling course. 0 stars. 0 forks. Static-Semantics-of-Cool-. This repository contains the static semantics of cool programming language developed as part of the programming assignment of Principle of Compiling course. About. This repository contains the static semantics of cool programming language developed as part of the programming assignment of Principle of Compiling course. Resources. Readme. Necula CS 164 Lecture 1715 Operational Rules of Cool The evaluation judgment is so, $E, S \vdash e : v, S'$. read: $\epsilon$ “Given so the current value of the self object $\epsilon$ And $E$ the current variable environment $\epsilon$ And $S$ the current store $\epsilon$ If the evaluation of $e$ terminates then $\epsilon$ “The returned value is $v$ $\epsilon$ And the new store is $S'$.” Necula CS 164 Lecture 1724 Operational Semantics of let Expressions (I) What is the context in which $e_2$ must be evaluated? $\epsilon$ “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 $v_1$ so, $E, S \vdash e_1 : v_1, S_1$ 1 so, $?, ?, e_2 : v, S_2$ so, $E, S \vdash \text{let } id : T \leftarrow e_1 \text{ in } e_2 : v_2, S_2$. The decomposition uses the structural operational semantics that underlies the process algebra. We use this decomposition method to derive congruence formats for two weak and rooted weak semantics: branching and \$\eta\$-bisimilarity. View. The weak transition systems obtained from such cool rules give rise to lax bialgebras, rather than to bialgebras.