Cool Operation Semantics

Greta Yorsh

Outline

- Motivation
- Notations
- Rules

Motivation

- Specify the semantics of every expression
  - what happens when an expression is evaluated
  - semantics = meaning

- Definition of a programming language:
  - regular expressions ⇒ lexical analysis
  - context free grammar ⇒ syntactic analysis
  - scoping rules ⇒ symbol table construction
  - typing rules ⇒ type cheching
  - **evaluation rules** ⇒ code generation and optimization

Evaluation rules

- So far, we specified the evaluation rules indirectly
  - first sections in Cool Manual
- We can specify evaluation rules by
  - compilation of Cool to MIPS (stack machine)
  - the evaluation rules of MIPS
- This is a **complete** description
- Why isn't it good enough?

How to specify evaluation rules?

- Specification should avoid **irrelevant details**
  - whether to use a stack machine or not
  - which way the stack grows
  - how integers are represented
  - the particular instruction set of the architecture
- Specification should be
  - complete
  - not overly restrictive

Programming language semantics

- Multiple powerful ways to specify semantics
- Suitable for different tasks

- Operational semantics
- Denotational semantics
- Axiomatic semantics
- …

Operational semantics

- Describes program evaluation steps
- Execution rules on an abstract machine

- Most useful for specifying implementations

- This is what we use for Cool

Denotational semantics

- Program's meaning is a **mathematical function**
- Need to define a suitable space of functions
- Foundation of programming language design

Axiomatic semantics

- Program behavior descried via logical formulas
  - Hoare logic { PRE } C { POST}
  - if execution of C begins in state satisfying PRE,
    then it ends in state satisfying POST
  - PRE, POST are logical formulas

- Foundation of many program **verification** systems
  - quick sort function sorts an array

Introduction to operational semantics

- Formal notation of inference rules
- Similar to type rules
- Typing judgment
  - **typeContext** $\vdash$ **e : T**
  - in the given **context**, expression **e** has type **T**
- Evaluation judgment
  - **evaluationContext** $\vdash$ **e : v**
  - in the given **context**, expression **e** evaluates to value **v**

Example: evaluation rules

- The result of evaluating an expression can depend on the result of evaluating its subexpressions
- The rules specify everything needed to evaluate an expression

$$\frac{\text{Context} \vdash e1 : 5 \qquad \text{Context} \vdash e2 : 7}{\text{Context} \vdash e1 + e2 : 12}$$

Context

- Keep track of values of variables
- Variables can change their values during the evaluation
- Example: $y \leftarrow x + 1$

- **Environment** the memory location where the value of a variable stored
- **Store** the contents of each memory location

Environment

- Maps variable names to memory locations
  - records where in memory the value of a variable is stored
  - keeps track of which variables are in scope

- **E = [x : l1, y : l2]**
- **E(x)** lookup a variable **x** in environment **E**

Store

- Maps memory locations to values

- **S = [l1 → 5, l2 → 7]**
- **S(l1)** lookup the contents of a **location l1** in **store S**
- **S' = S[12/l1]** defines a store S' such that
  - S'(l1) = 12   and
  - S'(l) = S(l)   if l ≠ l1
  - used to perform an <u>assignment of 12 to location l1</u>

Cool evaluation rules

- The evaluation judgment is **so, E, S** $\vdash$ **e : v, S'**
  - **so** the current value of the self object
  - **E** the current variable environment
  - **S** the current store
- If the evaluation of **e** terminates then
  - the returned value is **v**
  - the new store is **S'**

Result of evaluation

- Evaluating an expression
  - value
  - new store to model **side-effects**
  - variable environment does not change
  - value of “self” does not change
- Allows non-terminating evaluations

Variable references

- Lookup of variables
  - from name to location
  - from location to value
- The store does not change
- A special case for self:

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

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

Assignment

- Evaluate the right hand side to get a value and a new store S1
- Fetch the location of the assigned variable
- The result is the value v and an updated store
- The environment does not change

$$\frac{\begin{array}{l} so, E, S \vdash e : v, S1 \\ E(id) = l_{id} \\ S2 = S1[v/l_{id}] \end{array}}{so, E, S \vdash id \leftarrow e : v, S2}$$

Example

$E = [x : l_x, y : l_y]$

$S = [l_x \rightarrow Int(3), l_y \rightarrow Int(10)]$

$x \leftarrow y \leftarrow 2$

$$\frac{S_1 = S[Int(2)/l_y] = [\ l_x \rightarrow Int(3),\ l_y \rightarrow Int(2)\ ]}{so, E, S \vdash y \leftarrow 2 : Int(2), S_1}$$

$$\frac{S_2 = S_1[\ Int(2)\ /l_x] = [\ l_x \rightarrow Int(2),\ l_y \rightarrow Int(2)\ ]}{so, E, S \vdash x \leftarrow y \leftarrow 2 : Int(2), S_2}$$

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(a1 = l1, …, an = ln)**
  - **X** is the class of the object (its dynamic type)
  - **ai** are the attributes **including inherited** ones
  - **li** is the location where the value of ai is stored

Classes without attributes

- Special notation for cool values
  - Int(5) the integer 5
  - Bool(true) the boolean true
  - String(4, “Cool”) the string “Cool” of length 4

Base values

- The store does not change

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

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

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

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

Special value: void

- Member of all types
- Supports *isvoid* test
- No other operations can be performed on it
- Concrete implementations might use NULL here

Conditionals

- Evaluation sequence
  - e1 must be evaluated first to produce S1
  - then e2 can be evaluated to produce S2
- The result of evaluating e1 is a Bool object (why?)
- There is another, similar, rule for Bool(false)

$$\frac{\text{so, E, S} \vdash \text{e1 : Bool(true), S1} \qquad \text{so, E, S1} \vdash \text{e2 : v, S2}}{\text{so, E, S} \vdash \text{if e1 then e2 else e3 : v, S2}}$$

Sequential composition

- Evaluation sequence
- Only the last value is used
- But all the side-effects are collected

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

Loops

- If e1 evaluates to Bool(false) then the loop terminates immediately
- Use the side-effects from the evaluation of e1
- Result value void
- The typing rules ensure that e1 evaluates to a Bool object

$$\frac{so, E, S \vdash e1 : Bool(false), S1}{so, E, S \vdash \text{while } e1 \text{ loop } e2 \text{ pool} : void, S1}$$

Loops

- Evaluation sequence (S, S1, S2, S3)
- Evaluation of a loop is expressed in terms of the evaluation of itself in another state
- The result of evaluating e2 is discarded
- Only the side-effect is preserved

$$\frac{\begin{array}{l}\text{so, E, S} \vdash \text{e1 : Bool(true), S1} \\ \text{so, E, S1} \vdash \text{e2 : v, S2} \\ \text{so, E, S2} \vdash \text{while e1 loop e2 pool : void, S3}\end{array}}{\text{so, E, S} \vdash \text{while e1 loop e2 pool : void, S3}}$$

Let

- What is the context in which e2 must be evaluated?
- Environment like E but with a new binding of id to a fresh location lnew
- Store like S1 but with lnew mapped to v1

$$\frac{\begin{array}{l}\text{so}, E, S \vdash e1 : v1, S1 \\ \text{so}, ?, ? \vdash e2 : v, S2\end{array}}{\text{so}, E, S \vdash \text{let } id : T \leftarrow e1 \text{ in } e2 : v, S2}$$

Let

- lnew = newloc(S)
- lnew is a location that is not already used in S
- newloc is like dynamic memory allocation

$$\frac{\begin{array}{l}\text{so, E, S} \vdash \text{e1 : v1, S1} \\ \text{lnew = newloc(S1)} \\ \text{so, E[lnew/id] , S1[v1/lnew]} \vdash \text{e2 : v2, S2}\end{array}}{\text{so, E, S} \vdash \text{let id : T} \leftarrow \text{e1 in e2 : v2, S2}}$$

new T

- 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_{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
  $\text{class}(A) = (a_1 : T_1 \leftarrow e_1, \ldots, a_n : T_n \leftarrow e_n)$
  - a_i are the attributes (including inherited)
  - T_i are their declared types
  - e_i are the initializers

new T

$$
\frac{
\begin{array}{l}
T0 = \text{if } T == \text{SELF\_TYPE and } so = X(\ldots) \text{ then } X \text{ else } T \\
class(T0) = (a1 : T1 \leftarrow e1, \ldots, an : Tn \leftarrow en) \\
li = newloc(S) \text{ for } i = 1, \ldots, n \\
v = T0(a1 = l1, \ldots, an = ln) \\
E' = [a1 : l1, \ldots, an : ln] \\
S1 = S[D_{T1}/l1, \ldots, D_{Tn}/ln] \\
v, E', S1 \vdash \{ a1 \leftarrow e1; \ldots; an \leftarrow en; \} : vn, S2
\end{array}
}{
so, E, S \vdash \text{new } T : v, S2
}
$$

new T

- Observation: new SELF_TYPE allocates an object with the same dynamic type as self
- 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
    - starting value of attributes are the default ones
- The side-effect of initialization is preserved

Dispatch $\mathbf{e_0.f(e_1,\ldots,e_n)}$

- 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
- Find the definition of f for X
- Create n new locations
- Create an environment that maps formal arguments of f to those locations
- Initialize the locations with the actual arguments
- Set self to the target object
- Evaluate the body of f

More Notation

- For a class A and a method f of A
  impl(A, f) = (x_1, …, x_n, e_{body})
  - x_i are the names of the formal arguments
  - e_{body} is the body of the method
  - f can be inherited

Dispatch

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

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, except the implementation of f is taken from the specified class

Runtime errors in dispatch

- What happens if impl(X, f) is not defined?
- What happens if target object is void?

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

Runtime errors

- There are some runtime errors that the type checker does not try to prevent (can it ?)
  - dispatch on void
  - case on void
  - no matching branch in case
  - division by zero
  - substring out of range
  - heap overflow
- Execution must abort gracefully
  - with an error message not with segfault
- Operational rules do not cover these cases

Conclusions

- Cool operational rules are **precise and detailed**

- Most languages do not have a well specified operational semantics
- When portability is important an operational semantics becomes essential
- Notation we used for Cool is very limited