CSC 425 - Principles of Compiler Design I

Operational Semantics

Outline

- Operational semantics is a precise way of specifying how to evaluate a program
- A formal semantics tells you what each expression means
- Meaning depends on context: a variable environment will map variables to memory locations and a store will map memory locations to values

Motivation

- The meaning of an expression is what happens when it is evaluated
- 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 $\Rightarrow$ interpretation

Assembly Language Description of Semantics

- Assembly language descriptions of language implementation have too many irrelevant details
  - 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 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

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
  - The foundation of many program verification systems

Introduction to Operational Semantics

- Once again we introduce a formal notation using logical rules of inference
- Recall the typing judgement

  $$\mathit{Context} \vdash e : T$$

  (in the given *Context*, expression e has type T)
- We try something similar for evaluation

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

  (in the given *Context*, expression e evaluates to value v)

Example Operational Semantics Inference Rule

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

- 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?

- Contexts are needed to handle variables
- Consider the evaluation of `x := x + 1`
  - We need to keep track of values of variables
  - We need to allow variables to change their values during 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; locations = memory addresses
- Environment tracks in-scope variables only
- Example environment:

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

- To lookup a variable a in environment E, we write $E(a)$

Stores

- A store maps memory locations to values
- Example store:

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

- To lookup the contents of a location l_1 in store S, we write $S(l_1)$
- To perform an assignment of 23 to location l_1, we write $S[23/l_1]$; this denotes a new store S' such that $S'(l_1) = 23$ and $S'(l) = S(l)$ if $l \neq l_1$

Operational Rules

- The evaluation judgement is

  $$E, S \vdash e : v, S'$$

  read:

  - Given E the current 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 both a value and also a new store
- Changes to the store model side-effects, that is, assignments to mutable variables
- The variable environment does not change
- The operational semantics allows for non-terminating evaluations
- We define one rule for each kind of expression

Example Operational Semantics for Base Values

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

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

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

$$\frac{s \text{ is an string literal}}{E, S \vdash s : String(i), S}$$

- Note: no side effects in these cases
- Bool, Int, and String represent type constructors of some sort

Example Operational Semantics of Variable References

$$\frac{\begin{array}{c} E(id) = l_{id} \\ S(l_id) = v \end{array}}{E, S \vdash id : v, S}$$

- Note the double lookup of variables
  - First from name to location (compile time)
  - Then from location to value (run time)
- The store does not change

Example Operational Semantics of Assignment

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

- A three step process
  - Evaluate the right hand side; a value v 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

Example Operational Semantics of Conditionals

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

- The "threading" of the store enforces an evaluation sequence
  - e_1 must be evaluated first to produce S_1
  - The e_2 can be evaluated
- The result of evaluating e_1 is a boolean
  - The typing rules ensure this fact
  - There is another similar rule for $Bool(false)$

Example Operational Semantics of Sequences

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

- Again, the "threading" of the store enforces an evaluation sequence
- Only the last value is used
- But, all the side-effects are collected

Example Operational Semantics of Loops

$$\frac{E, S \vdash e_1 : Bool(false), S_1}{E, S \vdash while\ e_1\ do\ e_2 : void, S_1}$$

- If e_1 evaluates to $Bool(false)$, then the loop terminates immediately
  - With the side-effects from the evaluation of e_1
  - And with (arbitrary) result value *void*
- The typing rules ensure that e_1 evaluates to a boolean

Example Operational Semantics of Loops

$$\frac{\begin{array}{l} E, S \vdash e_1 : Bool(true), S_1 \\ E, S_1 \vdash e_2 : v, S_2 \\ E, S_2 \vdash while\ e_1\ do\ e_2 : void, S_3 \end{array}}{E, S \vdash while\ e_1\ do\ e_2 : void, S_3}$$

- 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 is preserved

Example Operational Semantics of Let Expressions

$$\frac{\begin{array}{c} E, S \vdash e_1 : v_1, S_1 \\ ?, ? \vdash e_2 : v, S_2 \end{array}}{E, S \vdash let\ id : T := e_1\ in\ e_2 : v_2, S_2}$$

- 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_{new}
  - Store like S_1, but with l_{new} mapped to v_1

Example Operational Semantics of Let Expressions

- We write $l_{new} = newloc(S)$ to say that l_{new} is a location that is not already used in S
  - Think of *newloc* as the dynamic memory allocation function (or reserving stack space)
- The operational rule for let:

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

Runtime Errors

- There are some runtime errors that the type checker does not try to prevent
  - Division by zero
  - Array out of bounds
  - Heap overflow
- In such cases, the execution must abort gracefully

Conclusions

- Operational rules are very precise; nothing is left unspecified
- Operational rules contain a lot of details
- Most languages do not have a well specified operational semantics
- When portability is important, an operational semantics becomes essential