

Lecture #28: Operational Semantics

- For syntax, we have BNF specifications of the proper *syntactic form* for programs, for which we have tools.
- For static semantics, we saw type specifications of what constitutes a *meaningful* program, for which we didn't have tools, but which give a complete and concise definition of the rules.
- Now we turn to *dynamic semantics*—the definition of what a program does or computes when executed.
- Again, we don't really have tools as we did for syntax, but a formal definition is helpful for defining the language precisely.

Approaches

- There are number of definitional methods.
- *Operational Semantics* in effect defines an abstract machine and translates each statement or expression into operations on that machine. (This is the one we'll use).
- *Denotational Semantics* gives a way of translating a program into a mathematical function on some domain that represents the state of a program.
- *Axiomatic Semantics* gives a way to translate a program interspersed with assertions about the state of that program (values of variables, etc.) into a mathematical assertion whose proof will indicate the correctness of that particular program relative to the assertions.

Operational Semantic Assertions

- Similarly to what we did for static semantics, we write assertions using a notation like this:

$$\frac{\vdots}{G, E, S \vdash e : v, S', R'}$$

where

- e is the language construct being defined,
- G, E, S embody the *evaluation context* before execution of e :
 - * G is the *global environment*, mapping *names* to *locations*.
 - * E is the *local environment*, also mapping names to locations.
 - * S is the *state* (of memory or *store*), mapping locations to values. Locations abstractions of *memory addresses*.
- v, S', R' embody the *result* of executing or evaluating e .
 - * v is *value* yielded by e (if any).
 - * S' is the state resulting from executing e .
 - * R' is the value returned by e (if it is a **return** statement).

Environments, Locations, and States

- Basic idea is that the store (or state) *contains the current values* manipulated by the program.
- Each value resides at a particular *location* in the store. We never say exactly what these are; they just come from some abstract set.
- That is, the store is a *function* mapping locations to values.
- Storing into a variable in memory will correspond to *replacing the state* with a new one.
- Environments map identifiers or names to locations.
- So “the value of x in environment E and state S ” translates to $S(E(x))$.
- And “the result of setting x to value v in environment E and state S ” is the new state $S[v/E(x)]$
- (Here, we use the same notation we used for indicating a change to an environment when discussing static semantics.)
- BTW: The same idea works for defining how arrays work (using indices in place of locations), or references (using pointers in place of locations and modeling the heap as a function like the state.)

Dynamic Semantic Rules

- Now that we have semantic assertions, we can use the same sort of notation for dynamic semantic rules as for static semantic type rules:

$$\frac{\textit{Hypotheses}}{G, E, S \vdash E : v, S', R'}$$

- Start with something really simple: **pass**

$$\frac{??}{G, E, S \vdash \text{pass} : _ , ?? , _} \quad [\text{PASS}]$$

- For this rule, the **pass** statement yields no value and does not cause a return, so we use '_' to indicate missing pieces.
- Actually, we never really use this rule in our code for this project, since we have removed all the **pass** statements during parsing.

Dynamic Semantic Rules

- Now that we have semantic assertions, we can use the same sort of notation for dynamic semantic rules as for static semantic type rules:

$$\frac{\textit{Hypotheses}}{G, E, S \vdash E : v, S', R'}$$

- Start with something really simple: **pass**

$$\frac{}{G, E, S \vdash \text{pass} : _, S, _} \quad [\text{PASS}]$$

- For this rule, the **pass** statement yields no value and does not cause a return, so we use '_' to indicate missing pieces.
- Actually, we never really use this rule in our code for this project, since we have removed all the **pass** statements during parsing.

Variables

- Reading (assigning) a variable involves finding its location in E and from that, yielding (modifying) its value in S' as indicated before.
- Here, the construct in question *does* produce a value (but of course, does not cause a return), and again does not change the state:

$$\frac{\begin{array}{l} E(id) = l_{id} \\ S(l_{id}) = v \end{array}}{G, E, S \vdash id : v, S, _} \quad [\text{VAR-READ}]$$

- Assignment, on the other hand, produces no value, but does produce a new state:

$$\frac{\begin{array}{l} G, E, S \vdash e : v, S_1, _ \\ E(id) = l_{id} \\ S_2 = S_1[v/l_{id}] \end{array}}{G, E, S \vdash id = e : _, S_2, _} \quad [\text{VAR-ASSIGN-STMT}]$$

Expression Statements

- An expression used as a statement is used only for its side-effects and has no value.

$$\frac{??}{G, E, S \vdash e : _, ??, _} \quad [\text{EXPR-STMT}]$$

Expression Statements

- An expression used as a statement is used only for its side-effects and has no value.

$$\frac{G, E, S \vdash e : v, S', _}{G, E, S \vdash e : _, S', _} \quad [\text{EXPR-STMT}]$$

A Word About Values

- For uniformity, the ChocoPy language reference treats all values as instances of classes.
- For a type T with attributes named a_1, \dots, a_n , a value of type T is denoted

$$T(a_1 = l_1, \dots, a_n = l_n)$$

- That is, every class value maps the attribute names into *locations* in the store that hold the values of those attributes.
- Why the indirection? Why not instead use the *values* of the attributes directly?
- The problem that locations solve is shown by examples like this:

```
class A(object):
    x: int = 3
anA: A = None
alias: A = None
anA = A()
alias = anA
anA.x = 4
```

Problem: How to explain that alias.x also changes.

Immutable

- The basic types `int`, `bool`, and `str` do not have mutable fields, so that it is unnecessary to use the indirection used for other classes.
- So the ChocoPy reference makes these special cases, and in the semantics, their values are represented instead as

`int(v)` # The `int` object representing the integer v .

`bool(b)` # The `bool` object representing True/False value b

`str(n, s)` # The `str` object representing the string s of length n

- Hence, we can write the rule for integer literals like this:

$$\frac{i \text{ is an integer literal}}{G, E, S \vdash i : \text{int}(i), S, _} \quad [\text{INT}]$$

(Well, strictly speaking, this is abuse of notation. The *numeral* i , which appears in the program, is the *denotation* of the *number*—the mathematical value i , so that in the last line of the rule, i means two different things. While I personally revel in such pedantry, it is perhaps not too important to be so fussy for the purposes of this course.)

Arithmetic

- When describing operations such as $e_1 + e_2$, we must take into account the fact that either e_1 or e_2 might modify the program state.
- Thus giving us this rule:

$$\frac{\begin{array}{l} G, E, S \vdash e_1 : \text{int}(i_1), S_1, - \\ G, E, S_1 \vdash e_2 : \text{int}(i_2), S_2, - \\ op \in \{+, -, *, //, \%\} \\ op \in \{//, \%\} \Rightarrow i_2 \neq 0 \\ v = \text{int}(i_1 \text{ op } i_2) \end{array}}{G, E, S \vdash e_1 \text{ op } e_2 : v, S_2, -} \quad [\text{ARITH}]$$

- There is a subtle point here: the above says that e_1 and e_2 must be evaluated in that order (why?).
- In contrast, the C language does not have this constraint, which gives compiler writers an easier time, but doesn't particularly help programmers and really complicates the formal semantics.