Lecture #18: Operational Semantics

Dynamic 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 to describe Chocopy).
- *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.

Axiomatic Semantics

- The idea is to describe the semantics of each kind of statement by *giving* axioms about what can be assumed about the state of the program after the statement is executed, *given* an assertion of what is true before the statement.
- The notation

$$\{P\}\ \mathcal{S}\ \{Q\}$$

  means "if P (a logical assertion) is true and statement $\mathcal{S}$ is executed, then Q will become true."
- Example: for any side-effect-free boolean expression $\mathcal{C}$, statements $\mathcal{S}_1$ and $\mathcal{S}_2$, and logical assertions P and Q, we may define the semantics of **if-else** as follows:

$$\frac{\begin{array}{c}\{P \wedge \mathcal{C}\}\ \mathcal{S}_1\ \{Q\} \\ \{P \wedge \sim \mathcal{C}\}\ \mathcal{S}_2\ \{Q\}\end{array}}{\{P\}\ \textbf{if}\ \mathcal{C}\ \textbf{then}\ \mathcal{S}_1\ \textbf{else}\ \mathcal{S}_2\ \textbf{fi}\{Q\}}$$

  That is, if the two assertions above the line are true, the assertion below the line will be true.
- We'll come back to this idea later in the semester.

Denotational Semantics

- With denotational semantics, we define the semantics of each kind of statement (and expression) by converting it into a function that takes the program state before its execution to (the resulting value and) the program state after its execution.
- Example: using $[\![\mathcal{S}]\!]$ to denote the function denoted by statement $\mathcal{S}$, we can define the semantics of sequencing with something like

$$[\![\mathcal{S}_1;\ \mathcal{S}_2]\!] = \lambda\sigma.[\![\mathcal{S}_2]\!]([\![\mathcal{S}_1]\!](\sigma))$$

  (where we use σ to denote program states).
- That is, the meaning of "$\mathcal{S}_1;\ \mathcal{S}_2$", which we write $[\![\mathcal{S}_1;\ \mathcal{S}_2]\!]$, is a function that applies $[\![\mathcal{S}_1]\!]$ (which is a function) to a program state σ, and then applies $[\![\mathcal{S}_2]\!]$ to the resulting state.
- But don't worry; we won't be going any further with denotational semantics this semester.

Operational Semantic Assertions for Chocopy

- 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 the *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 enviroment 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:

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

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

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

- 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:

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

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

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

- 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:

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

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

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

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 $a_1, \ldots, a_n$, a value of type T is denoted

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

- 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.
```

Immutables

- 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 repreenting 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:

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

(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:

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

- 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.

A Typo

- Suppose that the ChocoPy reference had this for the arithmetic rule instead (inspired by a typo there that was fixed some time ago):

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

- What would this mean?

A Typo

- Suppose that the ChocoPy reference had this for the arithmetic rule instead (inspired by a typo there that was fixed some time ago):

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

- What would this mean?
- It says that e_1 and e_2 must both have the same effect on the state for the rule to apply, and that they are both evaluated from the initial state. Definitely not what was intended!

Short-circuit Logical Operations

- The right operand of '**and**' is supposed to be evaluated if and only if the left operand yields True.
- Easy to do this with two rules that have mutually exclusive sets of hypotheses.

$$\frac{G, E, S \vdash e_1 : bool(false), S_1, \_}{G, E, S \vdash e_1 \text{ and } e_2 : bool(false), S_1, \_} \quad [\text{AND-}1]$$

$$\frac{\begin{array}{c} G, E, S \vdash e_1 : bool(true), S_1, \_ \\ G, E, S_1 \vdash e_2 : v, S_2, \_ \end{array}}{G, E, S \vdash e_1 \text{ and } e_2 : v, S_2, \_} \quad [\text{AND-}2]$$

- The AND-1 rule applies only if e_1 evaluates to false, and AND-2 applies only if e_1 evaluates to true.
- See if you can figure out the analogous rules for 'or'.

Returning

- Return statements have an interesting property: they must stop execution and propogate out of their enclosing statements.
- First, the return statement itself sets the R value in our assertions to something other than $\_$:

$$\frac{G, E, S \vdash e : v, S_1, \_}{G, E, S \vdash \texttt{return}\ e : \_, S_1, v} \quad [\text{RETURN-E}]$$

$$\frac{}{G, E, S \vdash \texttt{return} : \_, S, None} \quad [\text{RETURN}]$$

- Now we have to depict their effect on the surrounding program.
- We'll start with sequences of statements.

Statement Sequences

- A statement sequence is also executed for its side-effect alone.
- First, consider the case where none of the statements is or contains a **return** statement:

$$\frac{\begin{array}{c} n \geq 0 \\ ?? \end{array}}{G, E, S_0 \vdash s_1 \ \texttt{\textbackslash n}\ s_2 \ \texttt{\textbackslash n}\ \ldots\ s_n \ \texttt{\textbackslash n}\ : \_, ??, \_} \quad [\text{STMT-SEQ}]$$

(where $\texttt{\textbackslash n}$ is newline.)

Statement Sequences

- A statement sequence is also executed for its side-effect alone.
- First, consider the case where none of the statements is or contains a **return** statement:

$$\frac{\begin{array}{c} n \geq 0 \\ G, E, S_0 \vdash s_1 : \_, S_1, \_ \\ G, E, S_1 \vdash s_2 : \_, S_2, \_ \\ \vdots \\ G, E, S_{n-1} \vdash s_n : \_, S_n, \_ \end{array}}{G, E, S_0 \vdash s_1 \ \backslash\mathtt{n}\ s_2 \ \backslash\mathtt{n}\ \ldots\ s_n \ \backslash\mathtt{n}\ : \_, S_n, \_} \quad [\text{STMT-SEQ}]$$

(where $\backslash\mathtt{n}$ is newline.)

Statement Sequences With a Return

- But if statement k returns something, the statements starting at $k+1$ are irrelevant:

$$\frac{\begin{array}{c} n \geq 0 \\ G, E, S_0 \vdash s_1 : \_, S_1, \_ \\ G, E, S_1 \vdash s_2 : \_, S_2, \_ \\ \vdots \\ G, E, S_{k-1} \vdash s_k : \_, S_k, R \\ k \leq n, \qquad R \text{ is not } \_ \end{array}}{G, E, S_0 \vdash s_1 \ \backslash\mathtt{n}\ s_2 \ \backslash\mathtt{n}\ \ldots\ s_n \ \backslash\mathtt{n}\ : \_, S_k, R} \quad [\text{STMT-SEQ-RETURN}]$$

If Statements

- For conditional statements, can use the same trick as for 'and' and 'or': one rule for a true condition and one for false.
- We must be careful to make sure that any return values are propagated out of the statement.

$$\frac{\begin{array}{c} G, E, S \vdash e : bool(true), S_1, \_ \\ G, E, S_1 \vdash b_1 : \_, S_2, R \end{array}}{G, E, S \vdash \texttt{if}\ e\texttt{:}\ b_1\ \texttt{else:}\ b_2 : \_, S_2, R} \quad [\text{IF-ELSE-TRUE}]$$

$$\frac{\begin{array}{c} G, E, S \vdash e : bool(false), S_1, \_ \\ G, E, S_1 \vdash b_2 : \_, S_2, R \end{array}}{G, E, S \vdash \texttt{if}\ e\texttt{:}\ b_1\ \texttt{else:}\ b_2 : \_, S_2, R} \quad [\text{IF-ELSE-FALSE}]$$

- The use of R above causes any return value from the true or false branch to become the return value of the entire statement.

While Statements

- Again, we can use the same trick as for **if**, but how to get the effect of repetition without writing an infinite sequence of nested **if** statements??

$$\frac{??}{G, E, S \vdash \texttt{while}\ e\texttt{:}\ b : ??}\ [\text{WHILE}]$$

While Statements

- Ans: The **while** is really (tail-)recursive, so start with a base case:

$$\frac{G, E, S \vdash e : ??}{G, E, S \vdash \texttt{while}\ e\texttt{:}\ ??} \quad [\text{WHILE-1}]$$

While Statements

- The **while** is really (tail-)recursive, so start with a base case:

$$\frac{G, E, S \vdash e : bool(false), S_1, \_}{G, E, S \vdash \texttt{while}\ e\texttt{:}\ b : \_, S_1, \_} \quad [\text{WHILE-FALSE}]$$

- And then the inductive case:

While Statements

- The **while** is really (tail-)recursive, so start with a base case:

$$\frac{G, E, S \vdash e : bool(false), S_1, \_}{G, E, S \vdash \texttt{while}\ e\texttt{:}\ b : \_, S_1, \_} \quad [\text{WHILE-FALSE}]$$

- And then the inductive case:

$$\frac{\begin{array}{c} G, E, S \vdash e : bool(true), S_1, \_ \\ G, E, S_1 \vdash b : \_, S_2, \_ \\ G, E, S_2 \vdash \texttt{while}\ e\texttt{:}\ b : \_, S_3, R \end{array}}{G, E, S \vdash \texttt{while}\ e\texttt{:}\ b : \_, S_3, R} \quad [\text{WHILE-TRUE-LOOP}]$$

- What's missing?

While Statements

- The **while** is really (tail-)recursive, so start with a base case:

$$\frac{G, E, S \vdash e : bool(false), S_1, \_}{G, E, S \vdash \texttt{while}\ e\texttt{:}\ b : \_, S_1, \_} \quad [\text{WHILE-FALSE}]$$

- And then the inductive case:

$$\frac{\begin{array}{c} G, E, S \vdash e : bool(true), S_1, \_ \\ G, E, S_1 \vdash b : \_, S_2, \_ \\ G, E, S_2 \vdash \texttt{while}\ e\texttt{:}\ b : \_, S_3, R \end{array}}{G, E, S \vdash \texttt{while}\ e\texttt{:}\ b : \_, S_3, R} \quad [\text{WHILE-TRUE-LOOP}]$$

- Ans: And finally another base case:

$$\frac{\begin{array}{c} G, E, S \vdash e : bool(true), S_1, \_ \\ G, E, S_1 \vdash b : \_, S_2, R \\ R \text{ is not } \_ \end{array}}{G, E, S \vdash \texttt{while}\ e\texttt{:}\ b : \_, S_2, R} \quad [\text{WHILE-TRUE-RETURN}]$$

Allocating Variables

- How can we describe, mathematically, the allocation of space for variables?
- Creating a new variable evidently amounts to creating a new location that is currently not used.
- So we posit a function *newloc*, which is supposed to return such locations. But what paramaters does it need?
- Clearly, what *newloc* returns must depend on what's already in the store.
- The store is a function mapping locations to values, so "what's already in the store" is the *domain* of the store.
- Therefore, for store S, we'll write

$$newloc(S, n) \mapsto (l_1, \ldots, l_n),\ l_i \text{ distinct and } l_i \notin \text{domain}(S)$$

- And abbreviate $newloc(S) = newloc(S, 1)$.

Example: List Displays

- We'll represent lists as *sequences of locations*, $[l_0, \ldots, l_{n-1}]$, where location l_i is the location containing the value of element i of the list.

$$\frac{\begin{array}{c} n \geq 0 \\ G, E, S_0 \vdash e_1 : v_1, S_1, \_ \\ G, E, S_1 \vdash e_2 : v_2, S_2, \_ \\ \vdots \\ G, E, S_{n-1} \vdash e_n : v_n, S_n, \_ \\ l_1, \ldots, l_n = newloc(S_n, n) \\ v = [l_1, l_2, \ldots, l_n] \\ S_{n+1} = S_n[v_1/l_1][v_2/l_2]\ldots[v_n/l_n] \end{array}}{G, E, S_0 \vdash \texttt{[}\ e_1, e_2, \ldots, e_n\ \texttt{]}\ : v, S_{n+1}, \_} \quad [\text{LIST-DISPLAY}]$$

Operations on Lists

- Selection from and assignment to lists look like variable assignments (unsurprisingly):

$$\frac{\begin{array}{c} G, E, S_0 \vdash e_1 : v_1, S_1, \_ \\ G, E, S_1 \vdash e_2 : int(i), S_2, \_ \\ v_1 = [l_1, l_2, \ldots, l_n] \\ 0 \le i < n \\ v_2 = S_2(l_{i+1}) \end{array}}{G, E, S_0 \vdash e_1[e_2] : v_2, S_2, \_} \quad [\text{LIST-SELECT}]$$

$$\frac{\begin{array}{c} G, E, S_0 \vdash e_3 : v_r, S_1, \_ \\ G, E, S_1 \vdash e_1 : v_l, S_2, \_ \\ G, E, S_2 \vdash e_2 : int(i), S_3, \_ \\ v_l = [l_1, l_2, \ldots, l_n] \\ 0 \le i < n \\ S_4 = S_3[v_r / l_{i+1}] \end{array}}{G, E, S_0 \vdash e_1[e_2] = e_3 : \_, S_4, \_} \quad [\text{LIST-ASSIGN-STMT}]$$