Lecture #29: Operational Semantics, Part II

A Typo

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

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

- 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 recently fixed):

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

- 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 returns a value:

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

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

Statement Sequences

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

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

(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:

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

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??

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

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):

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

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

Operations on Lists: Concatenation

- List concatenation again requires allocation:

$$\frac{\begin{array}{c} G, E, S_0 \vdash e_1 : v_1, S_1, \_ \\ G, E, S_1 \vdash e_2 : v_2, S_2, \_ \\ v_1 = [l_1, l_2, \ldots, l_n] \\ v_2 = [l'_1, l'_2, \ldots, l'_m] \\ n, m \geq 0 \\ l''_1, \ldots, l''_{m+n} = newloc(S_2, m+n) \\ v_3 = [l''_1, l''_2, \ldots, l''_{n+m}] \\ S_3 = S_2[S_2(l_1)/l''_1] \ldots [S_2(l_n)/l''_n][S_2(l'_1)/l''_{n+1}] \ldots [S_2(l'_m)/l''_{n+m}] \end{array}}{G, E, S_0 \vdash e_1 \texttt{+} e_2 : v_3, S_3, \_} \quad [\text{LIST-CONCAT}]$$

- Subtlety here: in $e_1 + e_2$, suppose evaluating e_2 has a side-effect on e_1. What value goes into the resulting list?