CS164 Programming Languages and Compilers **Spring 2024**

Discussion Worksheet 8

1 Operational Semantics

Recall that judgements for Type Checking are of the form:

$$O, M, C \vdash e : T$$

Judgements for Operational Semantics are (for now) of the form:

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

Exercise 1 Let's begin by going over the basics of what operational semantics mean.

1.1 What are the values v?

Solution:
All values in ChocoPy are objects. We write $X(a_1 = l_1, a_2 = l_2, ...a_n = l_n)$ to mean the object with dynamic type X with attributes a_1 at location l_1, a_2 at location l_2, ..., and a_n at location l_n.

For convenience, we have abbreviations for the basic classes.

Bool(True), Bool(False) Int(0), Int(1), ... String(7, "ChocoPy"), ...

Finally, `None` is also a legal value.

1.2 How are these values related to types?

Solution: The static types we assign during type-checking should be a supertype of the dynamic type of the value.

1.3 What is the connection between the typing judgment and the operational semantics judgement? I.e., what does it mean for a program to be well-typed, and how does this relate to a program evaluating under operational semantics?

Solution:
Before when we talked about type soundness, we were not able to give a precise definition. We roughly said that type soundness means that if an expression is well-typed, then nothing can go bad at runtime. With operational semantics, we now have a precise definition of what "runtime" is. We now can say that our type system is sound with respect to the operational semantics if we can prove the following theorem.

if $O, M, C \vdash e : T$ and $E, S \vdash e : v, S'$ where $v = T'(...)$ then $T' \Leftarrow T$

That is, if the static type of an expression e is T, then the dynamic type of that expression is a subtype of T.

1.4 What is E?

Solution: The environment (E) is a mapping from variables to locations. You can think of locations as pointers. Essentially, the environment tells you which variables are defined, and where each variable is stored in memory.

1.5 What is S?

Solution: The store (S) is a mapping from locations to values. You can think of the store as the current contents of the computer's memory. When we say $S(l)$, we mean the value of location l in store S. If we think of locations as pointers, $S(l)$ is the value we get when dereferencing the pointer l.

Exercise 2 Let's consider the integer addition expression $e_1 + e_2$. This is the operational rule, along with the rule for assignment statements:

$$\text{ADD}\frac{\begin{array}{c} E, S_1 \vdash e_1 : int(i_1), S_2, \_ \\ E, S_2 \vdash e_2 : int(i_2), S_3, \_ \end{array}}{E, S_1 \vdash e_1 + e_2 : int(v), S_3, \_} \qquad \text{VAR-ASSIGNMENT-STMT}\frac{\begin{array}{c} E, S \vdash e : v, S_1, \_ \\ E(id) = l_{id} \\ S_2 = S_1[v/l_{id}]\_ \end{array}}{E, S \vdash id = e : \_, S_2, \_}$$

2.1 Why do we separate the store and the environment?

Solution: The environment tells us what variables are accessible. This is based on the static scoping rules. The store represents the memory at run time, which tells us the values of variables. It's convenient to separate these two concepts.
For example, separating these two concepts allows us to easily express variable aliasing. If we were writing semantics for a language where you can take the address of variables, the mapping from variables to locations would be quite useful, since it tells us where a variable is stored in memory.

2.2 Why does the add judgement update the store but not the environment?

Solution: The environment is only updated when new variables are introduced (e.g., in function calls), whereas the store is updated every time an assignment occurs.
Evaluating an expression cannot make permanent changes to the environment; the environment is only updated within a statically defined scope when a variable is introduced.
Changes to the store, however, are permanent; they have effects outside of their static scope. Thus, the judgment needs to yield an "updated" store that is used when evaluating subsequent expressions. (Check out the rule for assignments to see how the store is updated. Many other rules show how the store is "threaded" between several subexpressions.)

Exercise 3 Next, consider the if-else expression: b_1 `if` e `else` b_2.

3.1 Refresher: write the typing rule for the if-else expression.

$$\frac{\begin{array}{c} O, M, C, R \vdash e : bool \\ O, M, C, R \vdash b_1 : T_1 \\ O, M, C, R \vdash b_2 : T_2 \end{array}}{O, M, C, R \vdash b_1 \texttt{ if } e \texttt{ else } b_2 : T_1 \sqcup T_2}$$

3.2 Write the operational rules for the if-else expression.

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

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

3.3 Operational rules are more precise then typing rules. Why even bother with typing rules, then?

Solution: Following all the operational rules basically amount to executing the entire program. Typing rules are an over-approximation of the behavior of the program. But, typing rules provide an over-approximation that is useful to catch "obvious" typing bugs. Thus, typing rules allow us to check some key properties of the programs without having to execute the entire thing.

Exercise 4 Finally, we get to the purpose of the R in the conclusion of our operation semantics. The return statement has special, interruptive behavior, that is specially modeled in R .

Here are the operational rules for the `return` statement. Note how propagate the return value to R.

$$\frac{E, S \vdash e : v, S_1, \_}{E, S \vdash \texttt{return}\ \ e : \_, S_1, v}$$

$$\frac{}{E, S \vdash \texttt{return} : \_, S, \texttt{None}}$$

4.1 Recall the operational semantics of `while` shown in class, when the guard evaluates to `True`:

$$\text{WHILE-TRUE-LOOP}\frac{\begin{array}{c}E, S \vdash e_1 : bool(true), S_1, \_ \\ E, S_1 \vdash b_2 : \_, S_2, \_ \\ E, S_2 \vdash \texttt{while}\ \ e_1 \texttt{:} b_2 : \_, S_3, \_\end{array}}{E, S \vdash \texttt{while}\ \ e_1 \texttt{:} b_2 : \_, S_3, \_}$$

Note that these do not propagate or consider return values. Complete the operational semantics for the execution `while` when the body of the loop returns (WHILE-TRUE-LOOP). Then, update WHILE-TRUE-LOOP to propagate return values properly. Hint: For WHILE-TRUE-LOOP, support the case where the current body does not return, but it may in a subsequent iteration.

$$\text{WHILE-TRUE-RETURN}\frac{\begin{array}{c}E, S \vdash e_1 : bool(true), S_1, \_ \\ E, S_1 \vdash b_2 : \_, S_2, R \\ R \text{ is not } \_\end{array}}{E, S \vdash \texttt{while}\ \ e_1 \texttt{:} b_2 : \_, S_2, R}$$

$$\text{WHILE-TRUE-LOOP}\frac{\begin{array}{c}E, S \vdash e_1 : bool(true), S_1, \_ \\ E, S_1 \vdash b_2 : \_, S_2, \_ \\ E, S_2 \vdash \texttt{while}\ \ e_1 \texttt{:} b_2 : \_, S_3, R\end{array}}{E, S \vdash \texttt{while}\ \ e_1 \texttt{:} b_2 : \_, S_3, R}$$