Lecture #11: Typing Examples for ChocoPy

- Today, we'll adapt the notation from Tuesday to ChocoPy.
- In order to cover all constructs, we will need to augment the type environment with some other information.
- Our type assertions will have one of the forms

$$O, M, C, R \vdash e : T \qquad \textit{or} \qquad O, M, C, R \vdash s$$

depending on whether we are typing $\boxed{e}$xpressions or $\boxed{s}$tatements, where

  - O is a type environment as in the last lecture.
  - M is the member environment: $M(C, I)$ returns the type of the attribute or method named I in class C.
  - C is the enclosing class.
  - R is the type to be returned by the enclosing function or method.

- Hence the the type assertions above means

  The expression e type checks and has type T, or the construct s is correctly typed, given that O, M, C, R are the type environment, member environment, enclosing class, and expected return type.

Variable Access

- The type environment tells us about variables' types.

$$\frac{O(id) = T, \text{where } T \text{ is not a function type.}}{O, M, C, R \vdash id : T} \quad [\text{VAR-READ}]$$

- We only apply this rule when an identifier appears as an expression or the left side of an assignment.
- The provision that T not be a function type reflects the fact that in ChocoPy, functions are not first-class values. Their identifiers are handled elsewhere.

Variable Assignment

- Variable assignment is closely related:

$$\frac{\begin{array}{l} O(id) = T \\ O, M, C, R \vdash e_1 : T_1 \\ T_1 \leq_a T \end{array}}{O, M, C, R \vdash id = e_1} \quad [\text{VAR-ASSIGN-STMT}]$$

- An assignment is a statement, not an expression, so the final line lacks a ': T' annotation.
- We are making use of the $\leq_a$ relation between types: *assignment compatibility*.
- This is a slight tweak on the ChocoPy type hierarchy: $T_1 \leq_a T_2$ iff
  - $T_1 \leq T_2$ (i.e., ordinary subtyping).
  - T_1 is `<None>` and T_2 is not int, bool, or str.
  - T_2 is a list type $[T]$ and T_1 is `<Empty>`.
  - T_2 is the list type `[T]` and T_1 is `[<None>]`, where `<None>` $\leq_a T$.
- Here, `<Empty>` is the type of the empty list, and `<None>` is the type of `None`.

List Indexing and Assignment

- Selection is straightforward:

$$\frac{\begin{array}{c} O, M, C, R \vdash e_1 : [T] \\ O, M, C, R \vdash e_2 : int \end{array}}{O, M, C, R \vdash e_1[e_2] : T} \quad [\text{LIST-SELECT}]$$

- Assignments to list index expressions are really the same as for variables:

$$\frac{\begin{array}{l} O, M, C, R \vdash e_1 : [T_1] \\ O, M, C, R \vdash e_2 : int \\ O, M, C, R \vdash e_3 : T_3 \\ T_3 \leq_a T_1 \end{array}}{O, M, C, R \vdash e_1[e_2] = e_3} \quad [\text{LIST-ASSIGN-STMT}]$$

Variable Initialization

- Initialization is obviously related to assignment.

$$\frac{\begin{array}{l} O(id) = T \\ O, M, C, R \vdash e_1 : T_1 \\ T_1 \leq_a T \end{array}}{O, M, C, R \vdash id\colon\ T = e_1} \quad [\text{VAR-INIT}]$$

- This is a declaration and, like a statement, does not produce a value. The ':' here is part of ChocoPy syntax, and not part of a type rule.

Attributes (instance variables)

- The rules are closely related to VAR-READ and VAR-ASSIGN-STMT.
- But we refer to M to get the types.

$$\frac{\begin{array}{l} O, M, C, R \vdash e_0 : T_0 \\ M(T_0, id) = T \end{array}}{O, M, C, R \vdash e_0.id : T} \quad [\text{ATTR-READ}]$$

$$\frac{\begin{array}{l} M(C, id) = T \\ O, M, C, R \vdash e_1 : T_1 \\ T_1 \leq_a T \end{array}}{O, M, C, R \vdash id\colon\ T = e_1} \quad [\text{ATTR-INIT}]$$

Some Obvious Ones

$$\frac{}{O, M, C, R \vdash \texttt{pass}} \quad \text{[PASS]}$$

$$\frac{}{O, M, C, R \vdash \texttt{False} : bool} \quad \text{[BOOL-FALSE]}$$

$$\frac{}{O, M, C, R \vdash \texttt{True} : bool} \quad \text{[BOOL-TRUE]}$$

$$\frac{i \text{ is an integer literal}}{O, M, C, R \vdash i : int} \quad \text{[INT]}$$

$$\frac{s \text{ is a string literal}}{O, M, C, R \vdash s : str} \quad \text{[STR]}$$

$$\frac{}{O, M, C, R \vdash \texttt{None} : \texttt{<None>}} \quad \text{[NONE]}$$

$$\frac{}{O, M, C, R \vdash \texttt{[]} : \texttt{<Empty>}} \quad \text{[NIL]}$$

Some Binary Operators

$$\frac{\begin{array}{c} O, M, C, R \vdash e_1 : int \\ O, M, C, R \vdash e_2 : int \\ op \in \{+, -, *, //, \%\} \end{array}}{O, M, C, R \vdash e_1 \;op\;\; e_2 : int} \quad [\text{ARITH}]$$

$$\frac{\begin{array}{c} O, M, C, R \vdash e_1 : str \\ O, M, C, R \vdash e_2 : str \end{array}}{O, M, C, R \vdash e_1 + e_2 : str} \quad [\text{STR-CONCAT}]$$

- We do not deal with precedence rules here, but assume that parsing determines to what operands to apply these rules.
- The ARITH and STR-CONCAT rules illustrate that the hypotheses (above the line) determine the applicability of a rule to a given situation. So `3+2` is covered by the first rule, and `"Hello," + " world"` by the second.
- Neither rule says that (e.g.) `3 + "Hello"` is *illegal.*
- Instead, the point is that neither of them says it is *legal*, and in the absence of some applicable rule, type checking fails.

Using Least Upper Bounds: List Displays

- The empty list has a special type, assignable to other list types.

$$\frac{}{O, M, C, R \vdash \texttt{[]} : \texttt{<Empty>}} \quad \text{[NIL]}$$

- The type of list created by a non-empty display is the *least upper bound* (denoted $\sqcup$) of the types of its elements, where the relevant type relation is $\leq_a$, rather than pure subtype.

$$\frac{\begin{array}{l} n \geq 1 \\ O, M, C, R \vdash e_1 : T_1 \\ O, M, C, R \vdash e_2 : T_2 \\ \quad \vdots \\ O, M, C, R \vdash e_n : T_n \\ T = T_1 \sqcup T_2 \sqcup \ldots \sqcup T_n \end{array}}{O, M, C, R \vdash \texttt{[}e_1, e_2, \ldots, e_n\texttt{]} : \texttt{[}T\texttt{]}} \quad \text{[LIST-DISPLAY]}$$

- This rule causes apparent glitches:

```
x: [object] = None
x = [3, x]           # OK
x = [3]              # ERROR (why?)
```

Return

- The return statement is where the 'R' part of type rules comes in:

$$\frac{\begin{array}{l} O, M, C, R \vdash e : T \\ T \leq_a R \end{array}}{O, M, C, R \vdash \texttt{return}\ e} \quad [\text{RETURN-E}]$$

$$\frac{\texttt{<None>} \leq_a R}{O, M, C, R \vdash \texttt{return}} \quad [\text{RETURN}]$$

- The second rule forbids programs like this:

```
def f() -> int:
    return
```

  Since `None` may not be assigned to an `int` value.

- We don't deal here with an *implicit* return of `None` from a function returning `int`, as happens when there is no **return** statement along some path. Instead, we can deal with that by inserting a **return** at the end of any function with a path that does not contain a **return**.

Function Types

- The ChocoPy reference uses a somewhat nonstandard notation for function types in order to carry around a bit more information that's useful elsewhere.
- Here, I'll revise it a bit to make the traditional function type signature itself clear:

  $$\{T_1 \times T_2 \times \ldots \times T_n \to T_0;\ x_1, x_2, \ldots, x_n;\ v_1 : {T_1}', v_2 : {T_2}', \ldots, v_m : {T_m}'\}$$

  will denote a function whose

  - type is $T_1 \times T_2 \times \ldots \times T_n \to T_0$,
  - formal parameters names are x_i, and
  - local names (local variables and nested functions) are v_j with types ${T_j}'$.

Function Calls

$$\frac{\begin{array}{l} O, M, C, R \vdash e_1 : T''_1 \\ \quad\vdots \\ O, M, C, R \vdash e_n : T''_n \\ n \geq 0 \\ O(f) = \{T_1 \times \ldots \times T_n \rightarrow T_0;\ x_1, x_2, \ldots, x_n;\ v_1 : T_1{}', \ldots, v_m : T_m{}'\} \\ \forall 1 \leq i \leq n : T''_i \leq_a T_i \end{array}}{O, M, C, R \vdash f(e_1, e_2, \ldots, e_n) : T_0} \quad [\text{INVOKE}]$$

- Dispatching calls on class members are the same, except that we get the type from M rather than O:

$$\frac{\begin{array}{l} O, M, C, R \vdash e_1 : T''_1 \\ \quad\vdots \\ O, M, C, R \vdash e_n : T''_n \\ n \geq 1 \\ M(T''_1, f) = \{T_1 \times \ldots \times T_n \rightarrow T_0;\ x_1, x_2, \ldots, x_n;\ v_1 : T_1{}', \ldots, v_m : T_m{}'\} \\ T''_1 \leq_a T_1 \\ \forall 1 \leq 2 \leq n : T''_i \leq_a T_i \end{array}}{O, M, C, R \vdash e_1.f(e_2, \ldots, e_n) : T_0} \quad [\text{DISPATCH}]$$

Function Definition

$$T = \begin{cases} T_0, & \text{if -> is present,} \\ \texttt{<None>}, & \textit{otherwise.} \end{cases}$$

$$\frac{\begin{array}{l} O(f) = \{T_1 \times \ldots \times T_n \rightarrow T_0;\ x_1, x_2, \ldots, x_n;\ v_1 : T_1{}', \ldots, v_m : T_m{}'\} \\ n \geq 0 \qquad m \geq 0 \\ O[T_1/x_1] \ldots [T_n/x_n][T'_1/v_1] \ldots [T'_m/v_m], M, C, T \vdash b \end{array}}{O, M, C, R \vdash \texttt{def}\ f(x_1 \,{:}\, T_1, \ldots, x_n \,{:}\, T_n)\ [\![\texttt{->}\ T_0]\!]^? : b} \quad [\text{FUNC-DEF}]$$

- So the definition as a whole type checks if it gives the right types for parameters and locals, and...
- the body type checks after substituting the indicated types for the formal parameter and local variable and function names.
- Here, we finally do something other than passing the R parameter in. When type-checking the body, R becomes the return type, allowing us to type-check **return** statements correctly (see the last hypothesis).

Class and Method Definitions

- Here is where C comes into play:

$$\frac{O, M, C, R \vdash b}{O, M, \bot, R \vdash \texttt{class}\ C(S) : b} \quad [\text{CLASS-DEF}]$$

$$\frac{\begin{array}{l} T = \begin{cases} T_0, & \text{if -> is present,} \\ \texttt{<None>}, & \textit{otherwise.} \end{cases} \\ M(C, f) = \{T_1 \times \ldots \times T_n \rightarrow T;\ x_1, \ldots, x_n;\ v_1 : T'_1, \ldots, v_m : T'_m\} \\ n \geq 1 \qquad m \geq 0 \\ C = T_1 \\ O[T_1/x_1] \ldots [T_n/x_n][T'_1/v_1] \ldots [T'_m/v_m], M, C, T \vdash b \end{array}}{O, M, C, R \vdash \texttt{def}\ f(x_1 : T_1, \ldots, x_n : T_n)\ [\![\texttt{->}\ T_0]\!]^? : b} \quad [\text{METHOD-DEF}]$$

Getting Things Started

- Before applying these rules, we gather up definitions of variables, functions, and classes in order to get the initial O and M.
- Also, the global definitions are part of this initial O and M:

$$\begin{aligned}
O(len) &= \{object \rightarrow int;\ arg\} \\
O(print) &= \{object \rightarrow \texttt{<None>};\ arg\} \\
O(input) &= \{\rightarrow str\} \\
M(object, __init__) &= \{object \rightarrow \texttt{<None>};\ self\} \\
M(str, __init__) &= \{object \rightarrow \texttt{<None>};\ self\} \\
M(int, __init__) &= \{object \rightarrow \texttt{<None>};\ self\} \\
M(bool, __init__) &= \{object \rightarrow \texttt{<None>};\ self\}
\end{aligned}$$

- And the whole program is then

$$\frac{O, M, \perp, \perp \vdash program}{\vdash program}\ [\text{PROGRAM}]$$

where $\perp$ ("bottom") means something undefined.