CS164 Programming Languages and Compilers **Spring 2024**

Discussion Worksheet 5

1 Scopes and Environments

A major step in the semantic analysis of a ChocoPy program is to determine what each identifier in the program refers to. This is important both for typechecking, and later for code generation. In ChocoPy there are several different namespaces for identifiers: identifiers may be variable or function names, method names, attribute names, or type names.

Exercise 1 Consider the following ChocoPy program:

```
x : int = 5
class T(object):
    a : int = 10
    def m(self : T) -> int:
        return self.a + x
def f(y : T) -> int:
    def g(z : int) -> T:
        nonlocal y
        y = T()
        return y
    return x + g().m()
f(T())
```

1.1 Which identifiers refer to variables or functions, to methods, to attributes, or to types?

Variables: `x` and `y` and `z`

Functions: `f` and `g`

Methods: `m` (of class `T`)

Attributes: `a` (of class `T`)

Types: `T`. Also, `int` and `object` refer to types.

1.2 Suppose we replace the identifier `m` with `f` everywhere in this program. Does this change what any identifiers refer to? If yes, what are the changes? If no, why not?

Solution: This does not change what identifiers refer to; even though the last line is now `return x + g().f()`, there is no conflict between the two identifiers `f`; one is a method and one is a function, which are in different namespaces/environments. In partifular, one is in the variable/function environment O vs. the class environment M.

1.3 What are the contents of the type environment O at each of the following lines?

Line 5: $O =$ `{x:int, self:T, f:{T→int; y; g:{int→T;z}}}`

Line 8: $O =$ `{x:int, z:int, y:T, f:{T→int; y; g:{int→T;z}}}`

Line 11: $O =$ `{x:int, y:T, f:{T→int; y; g:{int→T;z}}}`

Line 12: $O =$ `{x:int, f:{T→int; y; g:{int→T;z}}}`

2 Typechecking

Recall that a type T_1 is assignable to T_2 (i.e. $T_1 \leq_a T_2$) if and only if one of the following holds:

- $T_1 \leq T_2$
- T_1 is `<None>` and T_2 is not one of `int`, `str`, or `bool`.
- T_1 is `<Empty>` and T_2 is $[T]$
- T_1 is `[<None>]` and T_2 is $[T]$, where `<None>` $\leq_a T$

Exercise 2 Consider the following ChocoPy program:

```
class D(object):
    d : int = 42
class E(object):
    e : int = 80
ds : [D] = None
es : [E] = None
ds = es = [None]
es[0] = E()
print(ds[0].d)
```

2.1 What "goes wrong" in the execution of program? Where?

Solution: At line 10, the attribute `d` will not exist for `ds[0]`, because `ds[0]` is actually an object of type `E`.

2.2 What is the underlying cause of this error?

Solution: The multiple assignment on line 7 made `ds` and `es` point to the same memory location, so a write of an object of type `E` to `es` caused that object to be added do `ds`, violating its type.

2.3 Say we want to prevent such errors through typechecking. What additional restrictions should we place on multiple assignment for it to typecheck? Do we need the same restriction on single assignment?

Solution: We need to place the restriction that `[None]` cannot be used on the RHS of a multiple assignment in order for it to typecheck. We don't need the same restriction on single assignment, since the variable it will be assigned to will have a non-`[None]` type and the assignment relation will prevent us from doing incompatible assignments.

2.4 We cannot use `<None>` in our type annotations. Suppose we *could*, so we could write:

```
{python}
empties : [None] = None
empties = [None]
```

Can you use `empties` in the above program to have a similar problem without using multiple assignment?
Solution: Yes we can. Consider:

```
empties : [None] = [None]
ds : [D] = empties
es : [E] = empties
ds[0] = D()
print(es[0].e)
```

Now let's take a look at how we can mechanically verify that a ChocoPy program typechecks, focusing on function invocation and lists. Recall the following typing rules:

$$\text{(INT)}\ \frac{i \text{ is an integer literal}}{O \vdash i : int}$$

$$\text{(NEW)}\ \frac{T \text{ is a class}}{O \vdash T() : T}$$

$$\text{(LIST-CONCAT)}\ \frac{O \vdash e_1 : [T_1] \qquad O \vdash e_2 : [T_2] \qquad T = T_1 \sqcup T_2}{O \vdash e_1 + e_2 : [T]}$$

$$\text{(LIST-DISPLAY)}\ \frac{O \vdash e_1 : T_1 \qquad O \vdash e_2 : T_2 \qquad \cdots \qquad O \vdash e_n : T_n \qquad T = T_1 \sqcup T_2 \sqcup \cdots \sqcup T_n}{O \vdash [e_1, e_2, \ldots, e_n] : [T]}$$

$$\text{(INVOKE)}\ \frac{\begin{array}{c} O \vdash e_1 : T_1'' \qquad O \vdash e_2 : T_2'' \qquad \cdots \qquad O \vdash e_2 : T_n'' \\ O(f) = \{T_1 \times \cdots T_n \rightarrow T_0; x_1, \ldots x_n; v_1 : T_1', \ldots v_m : T_m'\} \qquad \forall i, T_i'' \leq_a T_i \end{array}}{O \vdash f(e_1, e_2, \ldots, e_n) : T_0}$$

Exercise 3 Consider the following ChocoPy program:

```
{python}
class A(object):
    a : int = 42
class B(A):
    b : int = 80
class C(A):
    c : int = 20
```

```
{python}
def f() -> [B]
    return [B()]
def g(n: int) -> [B]:
    return [B()]
```

3.1 Using an inverted tree of typing judgments, prove that the expression `f() + [C()]` has type `[A]`.

Solution:

$$\text{(LIST-CONCAT)}\ \frac{\text{(INVOKE)}\ \dfrac{O(f) = \{\rightarrow \texttt{[B]}\}}{O \vdash \texttt{f()} : \texttt{[B]}} \qquad \text{(LIST-DISPLAY)}\ \dfrac{\text{(NEW)}\ \dfrac{}{O \vdash\ \texttt{C()} : \texttt{C}} \qquad C = C}{O \vdash\ \texttt{[C()]} : \texttt{[C]}} \qquad \texttt{A} = \texttt{B} \sqcup \texttt{C}}{O \vdash \texttt{f() + [C()]} : \texttt{[A]}}$$

3.2 Similarly, prove that the expression `g(1) + [10]` has type `[object]`.

Solution:

$$\text{(CONCAT)}\ \frac{\text{(INV.)}\ \dfrac{\text{(INT)}\ \dfrac{}{O \vdash \texttt{1} : \texttt{int}} \qquad O(g) = \{\texttt{int} \rightarrow \texttt{[B]}; n\} \qquad \texttt{int} \leq_a \texttt{int}}{O \vdash \texttt{g(1)} : \texttt{[B]}} \qquad \text{(LIST-DISP)}\ \dfrac{\text{(INT)}\ \dfrac{}{O \vdash\ \texttt{10} : \texttt{int}} \qquad \texttt{int} = \texttt{int}}{O \vdash \texttt{[10]} : \texttt{[int]}} \qquad \texttt{object} = \texttt{B} \sqcup \texttt{int}}{O \vdash \texttt{g(1) + [10]} : \texttt{[object]}}$$