CS131 Compilers: Writing Assignment 3
Due 11:59pm May 14, 2023

Name - ID

I worked with Name1 Name2 ...

Complteted on May 3, 2023

Code of Conduct

This writing assignments should be your own individual work. Discussion on concept, methodology, and class materials are welcomed, but you should list all the people you have discussed with. Copying is strictly prohibited. Plagiarism, once confirmed, may result in assignment grades reduced to zero for all involved people. And this event will be reported. Also you should use LaTeX or Typst to produce your response based on this template. Submission in other forms won't be graded.

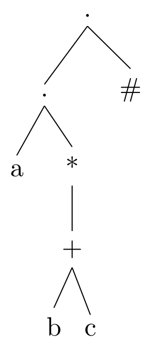

Figure 1: Simple Tree drawn by Tikz-qtree package

$$\dfrac{\dfrac{\dfrac{A}{B}\quad C}{D}\qquad \dfrac{\dfrac{E\quad F}{G}}{H}}{J}$$

Figure 2: An inference tree drawn by ebproof package. Google it for more tutorials

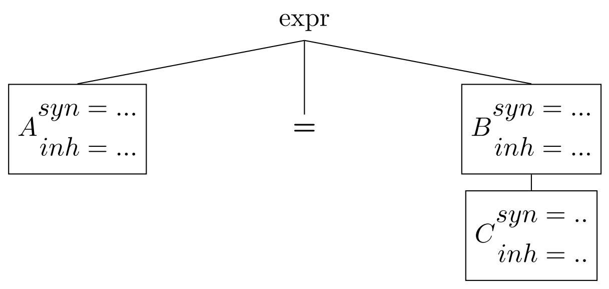
expr
$A^{syn = ...}_{inh = ...}$
=
$B^{syn = ...}_{inh = ...}$
$C^{syn = ..}_{inh = ..}$

Figure 3: A simple SSD tree Drawn by Tikz package

1. (**3 + 3 + 3 + 3 = 12 pts**) Consider this snippet of code, give out its corresponding...

```
t1 = t1 * t2 + t3 * (-t2)
```

   (a) Three Address Code
   (b) Quadruples Table
   (c) Triples Table
   (d) Indirect Triples Table
   RESPONSE:

2. (**10 pts**) Consider the Syntax-Directed Definition below, which is simplified from Chocopy.

D → TL	L.inh = T.type
T → int	T.type = integer
T → float	T.type = float
L → L_1, id	L_1.inh = L.inh
	addType(id.entry, L.inh)
L_1 → id	addType(id.entry, L.inh)

Draw the annotated parse tree of the corresponding snippets of code.

```
float a,b,c
```

RESPONSE:

3. (**5 + 10 + 15 = 30 pts**)Read the ChocoPy type rules (which has been appended at the end of file) and the code snippets. Complete the following task.

```
1  class A(object):
2      aA:int = 1
3  class B(A):
4      bB:bool = False
5
6  a_class: A = None
7  b_class: B = None
8
9  a:int = 1
10
11 arrA: [A] = None
12 arrB: [B] = None
13
14 def Func1(aa:int) -> int:
15     return aa + 1
16
17 arrB = [b_class]
18 arrA = [a_class] + arrB
```

(a) What's the meaning of O, M, C, R respectively in the ChocoPy type rules.
RESPONSE:

(b) At line 15, what does O, M, C, R evaluate to respectively.
RESPONSE:

(c) Prove line 18 is a valid statement by demonstrating that the type checking rule [var-assign-stmt] holds.
RESPONSE:

End of This Assignment

below is the type rules of ChocoPy

5 Type rules

This section formally defines the type rules of ChocoPy. The type rules define the type of every ChocoPy expression in a given context. The context is the type environment, which describes the type of every unbound identifier appearing in an expression. The type environment is described in Section 5.1. Section 5.2 gives the type rules.

5.1 Type environments

To a first approximation, type checking in ChocoPy can be thought of as a bottom-up algorithm: the type of an expression e is inferred from the (previously inferred) types of e's sub-expression. For example, an integer `1` has type `int`; there are no sub-expression in this case. As another example, if the types of e_1 and e_2 are `int`, then the expression $e_1 > e_2$ has type `bool`.

A complication arises in the case of an expression v, where v is a variable or a function. It is not possible to say what the type of v is in a strictly bottom-up algorithm; we need to know the type declared for v in the larger expression. Such a declaration must exist for every variable and function in valid ChocoPy programs.

To capture information about the types of identifiers, we use a *type environment*. The type environment consists of four parts: O a local environment, M a method/attribute environment M, C the name of the current class in which the expression or statement appears, and R the return type of the function or method in which the expression or statement appears. C is $\bot$ when the expression or statement appears outside a class, i.e. as a statement or expression in the top level. Similarly, R is $\bot$ when the expression or statement appears outside a function or method, i.e. as a statement or expression in the top level. The local environment and the method/attribute environment are both maps. The local environment is a function of the form:

$$O(v) = T$$

which assigns the type T to a variable v. The same environment also holds information about function signatures. For example,

$$O(f) = \{T_1 \times \cdots \times T_n \rightarrow T_0;\ x_1, \ldots, x_n;\ v_1 : T'_1, \ldots, v_m : T'_m\}$$

gives the type of f and denotes that identifier f has formal parameters $x_1, \ldots x_n$ of types $T_1, \ldots, T_n$, respectively, and has return type T_0. The identifiers $v_1, \ldots, v_m$ are the variables and nested functions declared in the body of f and their types are $T'_1, \ldots, T'_m$, respectively. The method/attribute environment similarly maps a class and its attributes and methods to their types. For example,

$$M(C, a) = T$$

maps the attribute a in the class C to the type T. Similarly,

$$M(C, m) = \{T_1 \times \cdots \times T_n \rightarrow T_0;\ x_1, \ldots, x_n;\ v_1 : T'_1, \ldots, v_k : T'_k\}$$

maps the method m of class C to it type. Specifically, it denotes that method m in class C has formal parameters $x_1, \ldots x_n$ of types $T_1, \ldots, T_n$, respectively, and has return type T_0. The identifiers $v_1, \ldots, v_k$ are the variables and nested functions declared in the body of m and their types are $T'_1, \ldots, T'_k$, respectively.

The third component of the type environment is the name of the class containing the expression or statement to be type checked. The fourth component of the type environment is the return type R of the function or method containing the expression or statement to be type checked.

When type checking function and method definitions, we need to propagate the typing environment from an outer scope to the function scope, where the binding of any identifier is inherited unless the function declares a formal parameter, variable, or a nested function with the same name. Let O be the current local environment and f be a function with type definition $\{T_1 \times \cdots \times T_n \rightarrow T_0;\ x_1, \ldots, x_n;\ v_1 : T'_1, \ldots, v_m : T'_m\}$. When type checking the definition of f, we type check its body using the local environment $O[T_1/x_1][T_2/x_2] \ldots [T_n/x_n][T'_1/v_1] \ldots [T'_m/v_m]$, where the notation $O[T/c]$ is used to construct a new mapping as follows:

$$O[T/c](c) = T$$
$$O[T/c](d) = O(d) \text{ if } d \neq c$$

5.2 Type checking rules

The general form of a type checking rule is:

$$\frac{\vdots}{O, M, C, R \vdash e : T}$$

The rule should be read: in the type environment with local environment O, method/attribute environment M, containing class C, and return type R, the expression e has type T. The line below the bar is a typing judgment: the turnstyle "$\vdash$" separates context (O, M, C, R) from a proposition $e : T$. The dots above the horizontal bar stand for other judgments about the types of sub-expressions of e. These other judgments are hypotheses of the rule; if the hypotheses are satisfied, then the judgment below the bar is true.

Variables. The rule for variables is simply that if the environment assigns an identifier *id* a type T, then the expression *id* has type T.

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

We must, however, prohibit identifiers with function types when reading values (that is, when identifiers are used as expressions in the syntax). This simply reflects the fact that ChocoPy does not treat functions as first-class (assignable, storable) values.

Variable Definitions and Assignments. This assignment rule—as well as others—uses the relation $\leq_a$ (ref. Section 2.4). The rule says that the assigned expression e_1 must have a type T_1 that is assignment compatible with the type T of the identifier *id* in the type environment.

$$\frac{\begin{array}{l} O(id) = T \\ O, M, C, R \vdash e_1 : T_1 \\ T_1 \leq_a T \end{array}}{O, M, C, R \vdash id = e_1} \quad [\textsc{var-assign-stmt}]$$

Variable definitions obey a similar rule:

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

The colon used below the line in the rule for VAR-INIT is the colon in the syntax for type annotations.

Statement and Definition Lists. These type check if all the component definitions and statements type check.

$$\frac{\begin{array}{l} O, M, C, R \vdash s_1 \\ O, M, C, R \vdash s_2 \\ \vdots \\ O, M, C, R \vdash s_n \\ n \geq 1 \end{array}}{O, M, C, R \vdash s_1 \text{ NEWLINE } s_2 \text{ NEWLINE } \ldots\ s_n \text{ NEWLINE}} \quad [\text{STMT-DEF-LIST}]$$

Pass Statements.

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

Expression Statements.

$$\frac{O, M, C, R \vdash e : T}{O, M, C, R \vdash e} \quad [\text{EXPR-STMT}]$$

Literals.

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

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

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

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

The `None` literal is assigned the (unmentionable) type `<None>`:

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

Arithmetic and Numerical Relational Operators.

$$\frac{O, M, C, R \vdash e : \mathit{int}}{O, M, C, R \vdash \texttt{-}\ e : \mathit{int}} \quad [\text{NEGATE}]$$

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

$$\frac{\begin{array}{l} O, M, C, R \vdash e_1 : \mathit{int} \\ O, M, C, R \vdash e_2 : \mathit{int} \\ \bowtie\ \in \{\texttt{<}, \texttt{<=}, \texttt{>}, \texttt{>=}, \texttt{==}, \texttt{!=}\} \end{array}}{O, M, C, R \vdash e_1 \bowtie e_2 : \mathit{bool}} \quad [\text{INT-COMPARE}]$$

Logical Operators.

$$\frac{\begin{array}{c}O, M, C, R \vdash e_1 : \mathit{bool} \\ O, M, C, R \vdash e_2 : \mathit{bool} \\ \bowtie \in \{\texttt{==}, \texttt{!=}\}\end{array}}{O, M, C, R \vdash e_1 \bowtie e_2 : \mathit{bool}} \quad [\textsc{bool-compare}]$$

$$\frac{\begin{array}{c}O, M, C, R \vdash e_1 : \mathit{bool} \\ O, M, C, R \vdash e_2 : \mathit{bool}\end{array}}{O, M, C, R \vdash e_1 \texttt{ and } e_2 : \mathit{bool}} \quad [\textsc{and}]$$

$$\frac{\begin{array}{c}O, M, C, R \vdash e_1 : \mathit{bool} \\ O, M, C, R \vdash e_2 : \mathit{bool}\end{array}}{O, M, C, R \vdash e_1 \texttt{ or } e_2 : \mathit{bool}} \quad [\textsc{or}]$$

$$\frac{O, M, C, R \vdash e : \mathit{bool}}{O, M, C, R \vdash \texttt{not } e : \mathit{bool}} \quad [\textsc{not}]$$

Conditional Expressions.

$$\frac{\begin{array}{c}O, M, C, R \vdash e_0 : \mathit{bool} \\ O, M, C, R \vdash e_1 : T_1 \\ O, M, C, R \vdash e_2 : T_2\end{array}}{O, M, C, R \vdash e_1 \texttt{ if } e_0 \texttt{ else } e_2 : T_1 \sqcup T_2)} \quad [\textsc{cond}]$$

String Operations.

$$\frac{\begin{array}{c}O, M, C, R \vdash e_1 : \mathit{str} \\ O, M, C, R \vdash e_2 : \mathit{str} \\ \bowtie \in \{\texttt{==}, \texttt{!=}\}\end{array}}{O, M, C, R \vdash e_1 \bowtie e_2 : \mathit{bool}} \quad [\textsc{str-compare}]$$

$$\frac{\begin{array}{c}O, M, C, R \vdash e_1 : \mathit{str} \\ O, M, C, R \vdash e_2 : \mathit{str}\end{array}}{O, M, C, R \vdash e_1 \texttt{ + } e_2 : \mathit{str}} \quad [\textsc{str-concat}]$$

$$\frac{\begin{array}{c}O, M, C, R \vdash e_1 : \mathit{str} \\ O, M, C, R \vdash e_2 : \mathit{int}\end{array}}{O, M, C, R \vdash e_1[e_2] : \mathit{str}} \quad [\textsc{str-select}]$$

The 'is' Operator.

$$\frac{\begin{array}{l}O, M, C, R \vdash e_1 : T_1 \\ O, M, C, R \vdash e_2 : T_2 \\ T_1, T_2 \text{ are not one of } \mathit{int}, \mathit{str}, \mathit{bool}\end{array}}{O, M, C, R \vdash e_1 \texttt{ is } e_2 : \mathit{bool}} \quad [\textsc{is}]$$

Object Construction. If T is the name of a class, then object-construction expressions of that class can be typed as follows:

$$\frac{T \text{ is a class}}{O, M, C, R \vdash T() : T} \quad [\textsc{new}]$$

List Displays.

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

The empty list is a special case:

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

List Operators.

$$\frac{\begin{array}{l} O, M, C, R \vdash e_1 : [T_1] \\ O, M, C, R \vdash e_2 : [T_2] \\ T = T_1 \sqcup T_2 \end{array}}{O, M, C, R \vdash e_1 \texttt{ + } e_2 : [T]} \quad [\textsc{list-concat}]$$

$$\frac{\begin{array}{l} O, M, C, R \vdash e_1 : [T] \\ O, M, C, R \vdash e_2 : int \end{array}}{O, M, C, R \vdash e_1[e_2] : T} \quad [\textsc{list-select}]$$

$$\frac{\begin{array}{l} O, M, C, R \vdash e_1 : [T_1] \\ O, M, C, R \vdash e_2 : int \\ O, M, C, R \vdash e_3 : T_3 \\ T_3 \leq_a T_1 \end{array}}{O, M, C, R \vdash e_1[e_2] = e_3} \quad [\textsc{list-assign-stmt}]$$

Attribute Access, Assignment, and Initialization. For attribute access, we use the class-member environment M:

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

$$\frac{\begin{array}{l} O, M, C, R \vdash e_0 : T_0 \\ O, M, C, R \vdash e_1 : T_1 \\ M(T_0, id) = T \\ T_1 \leq_a T \end{array}}{O, M, C, R \vdash e_0.id = e_1} \quad [\textsc{attr-assign-stmt}]$$

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

Multiple Assignments. Multiple assignment is type-checked by decomposing into individual single assignments, as follows.

$$\frac{\begin{array}{l} n > 1 \\ O, M, C, R \vdash e_0 : T_0 \\ O, M, C, R \vdash e_1 = e_0 \\ \vdots \\ O, M, C, R \vdash e_n = e_0 \\ T_0 \neq \texttt{[<None>]} \end{array}}{O, M, C, R \vdash e_1 = e_2 = \cdots = e_n = e_0} \quad [\text{MULTI-ASSIGN-STMT}]$$

The restriction that $T_0 \neq$ `[<None>]` avoids a subtle type-safety issue. It is dangerous to allow there to be two different views of a list with differing element types. The type `[<None>]` can only arise from list displays. As long as the value of such a display is immediately consumed by assignment to a single variable, parameter, or operand (`+` for lists), there will be only be one opinion as to its type subsequently. But multiple assignment opens the possibility of programs like this:

```
x: [A] = None
y: [[int]] = None
x = y = [None]      # Trouble ahead!
x[0] = A()
print(y[0][0])      # ???
```

Java, for example, gets itself into this bind and therefore needs a runtime `ArrayStoreException`. To avoid it in ChocoPy, we conservatively forbid values of the type `[<None>]` to be multiply assigned.
There is another very subtle point lurking here in the case where e_0 has type `<Empty>` (the type of the empty list). In this case, however, we need no special rule because in ChocoPy, there is no `.append` method to allow elements to be added to an empty list. Were that not the case, we could get this situation:

```
A: [int] = None
B: [str] = None
A = B = []
A.append(3)
```

and we'd subsequently have `B[0]` returning the value 3, which is certainly not a string.

Function Applications.

$$\frac{\begin{array}{l} O, M, C, R \vdash e_1 : T_1'' \\ O, M, C, R \vdash e_2 : T_2'' \\ \vdots \\ O, M, C, R \vdash e_n : T_n'' \\ n \geq 0 \\ O(f) = \{T_1 \times \cdots \times T_n \rightarrow T_0;\ x_1, \ldots, x_n;\ v_1 : T_1', \ldots, v_m : T_m'\} \\ \forall 1 \leq i \leq n : T_i'' \leq_a T_i \end{array}}{O, M, C, R \vdash f(e_1, e_2, \ldots, e_n) : T_0} \quad [\text{INVOKE}]$$

To type check a function invocation, each of the arguments to the function must be first type checked. The type of each argument must conform to the types of the corresponding formal parameter of the function. The invocation expression is assigned the function's declared return type.

Method dispatch expressions are type checked in a similar fashion. The key difference from the function invocation expression is that the target method is determined by consulting the method/attribute environment using the type of the receiver object expression:

$$\frac{\begin{array}{l} O, M, C, R \vdash e_1 : T_1'' \\ O, M, C, R \vdash e_2 : T_2'' \\ \vdots \\ O, M, C, R \vdash e_n : T_n'' \\ n \geq 1 \\ M(T_1'', f) = \{T_1 \times \cdots \times T_n \to T_0;\ x_1, \ldots, x_n;\ v_1 : T_1', \ldots, v_m : T_m'\} \\ T_1'' \leq_a T_1 \\ \forall i.\ 2 \leq i \leq n : T_i'' \leq_a T_i \end{array}}{O, M, C, R \vdash e_1.f(e_2, \ldots, e_n) : T_0} \quad [\textsc{dispatch}]$$

Return Statements. This is where the return-type environment comes into play:

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

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

Conditional Statements.

$$\frac{\begin{array}{l} O, M, C, R \vdash e_0 : bool \\ O, M, C, R \vdash b_0 \\ O, M, C, R \vdash e_1 : bool \\ O, M, C, R \vdash b_1 \\ \vdots \\ O, M, C, R \vdash e_n : bool \\ O, M, C, R \vdash b_n \\ n \geq 0 \\ O, M, C, R \vdash b_{n+1} \end{array}}{O, M, C, R \vdash \texttt{if}\ e_0 \texttt{:} b_0\ \texttt{elif}\ e_1 \texttt{:} b_1 \ldots\ \texttt{elif}\ e_n \texttt{:} b_n\ \texttt{else:} b_{n+1}} \quad [\textsc{if-elif-else}]$$

While Statements.

$$\frac{\begin{array}{l} O, M, C, R \vdash e : bool \\ O, M, C, R \vdash b \end{array}}{O, M, C, R \vdash \texttt{while}\ e \texttt{:} b} \quad [\textsc{while}]$$

For Statements.

$$\frac{\begin{array}{l} O, M, C, R \vdash e : str \\ O(id) = T \\ str \leq_a T \\ O, M, C, R \vdash b \end{array}}{O, M, C, R \vdash \texttt{for}\ id\ \texttt{in}\ e \texttt{:} b} \quad [\textsc{for-str}]$$

$$\frac{\begin{array}{l} O, M, C, R \vdash e : [T_1] \\ O(id) = T \\ T_1 \leq_a T \\ O, M, C, R \vdash b \end{array}}{O, M, C, R \vdash \texttt{for}\ id\ \texttt{in}\ e \texttt{:} b} \quad [\text{FOR-LIST}]$$

Function Definitions. To type a function definition for f, we check the body of the function f in an environment where O is extended with bindings for the names explicitly declared by f.

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

Likewise for methods:

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

Class Definitions. Class definitions are type checked by propagating the appropriate typing environment:

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

The Global Typing Environment. The following functions and class methods are predefined globally:

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

6 Operational semantics

This section contains the formal operational semantics for the ChocoPy language. The operational semantics define how every definition, statement, or expression in a ChocoPy program should be evaluated in a given context. The context has four components: a global environment, a local environment, a store, and a return