CS164 Programming Languages and Compilers **Spring 2024**

Written Assignment 5

Assigned: April 1 **Due:** April 11 at 11:59pm

Instructions: This assignment asks you to prepare written answers to questions on operational semantics and on garbage collection. Each of the questions has a short answer. You may discuss this assignment with other students and work on the problems together. However, your write-up should be your own individual work.

Please write your name, email address, and discussion section on your homework. ***Please start each question on a new page. All written assignments must be submitted as a PDF via Gradescope:*** *https://gradescope.com.* Instructions for how to submit assignments to Gradescope can be found at the following links: `https://gradescope.com/get_started#student-submission`

1. Consider the following ChocoPy program:

```
x:int = 1
class A(object):
    a1:int = 0
    a2:int = 1
    def __init__(self:"A"):
        global x
        x = x + 1

    def foo(self:"A", x:int) -> int:
        def bar() -> int:
            return x
        return bar()
bar : A = None
bar = A()
bar.a1
```

(a) Give an inverted evaluation tree, using operational semantics judgments, for the right side of the assignment on the second to last line: `A()`. You may assume that the environments and store G, E, and S are available. We have provided a template for the inverted tree, leaving blanks in lemmas (i), (ii), and (iii) for you to fill out.

$class(A) = (a1 = 0, a2 = 1, \_\_init\_\_ = \texttt{def \_\_init\_\_(self : "A")},$
$\quad foo = \texttt{def foo(self : "A", x : int)} \rightarrow \texttt{int})$
$l_{a1}, l_{a2}, l_{\_\_init\_\_}, l_{foo} = newloc(S, 4)$
$v_0 = A(a1 = l_{a1}, a2 = l_{a2}, \_\_init\_\_ = l_{\_\_init\_\_}, foo = l_{foo})$
$G, G, S \vdash 0 : int(0), S, \_$
$G, G, S \vdash 1 : int(1), S, \_$
$G, G, S \vdash \texttt{def \_\_init\_\_(self : "A")}\{...\} : v_{\_\_init\_\_}, S, \_$ from lemma (i)
$G, G, S \vdash \texttt{def foo(self : "A", x : int)} \rightarrow \texttt{int}\{...\} : v_{foo}, S, \_$ from lemma (ii)
$S_1 = S[int(0)/l_{a1}][int(1)/l_{a2}][v_{\_\_init\_\_}/l_{\_\_init\_\_}][v_{foo}/l_{foo}]$
$S_1(l_{init}) = (self, \texttt{x} = \texttt{x} + \texttt{1}, E_{\_\_init\_\_})$
$l_{self} = newloc(S_1)$
$E' = E_{\_\_init\_\_}[l_{self}/self]$
$S_2 = S_1[v_0/l_{self}]$
$G, E', S_2 \vdash \texttt{x} = \texttt{x} + \texttt{1} : \_, S_3, \_$ from lemma (iii)

$G, E, S \vdash A() : v_0, S_3, \_$

(i)

x declared to be global in __init__
$E_{\_\_init\_\_} =$
$v_{\_\_init\_\_} =$

$G, G, S \vdash \texttt{def \_\_init\_\_(self : "A")}\{...\} : v_{\_\_init\_\_}, S, \_$

(ii)

$bar =$ `def bar()` $\rightarrow$ `int` is a nested function defined in foo
$E_{foo} =$
$v_{foo} =$

$G, G, S \vdash$ `def foo(self :` "`A`"`, x : int)` $\rightarrow$ `int{...}` $: v_{foo}, S, \_$

(iii)

$E'(x) =$
$S_2(l_x) =$

$G, E', S_2 \vdash$ `x` $: int(1), S_2, \_$
$G, E', S_2 \vdash 1 : int(1), S_2, \_$
$1 + 1 = 2$

$G, E', S_2 \vdash$ `x + 1` $: int(2), S_2, \_$
$E'(x) =$
$S_3 =$

$G, E', S_2 \vdash$ `x = x + 1` $: \_, S_3, \_$

(b) Give an inverted evaluation tree, using operational semantics judgments, for the last line: `bar.a1`. You may assume as before that the environments and store G, E, and S are available.

2. Consider the following ChocoPy program:

```
x:int = 0
y:int = 0
def f1() -> int:
    global y
    def f2() -> int:
        return x + 1
    y = 2
    return f2()
f1()
```

Assume that the environments and store G, E, and S are available. Write the inverted evaluation tree for the expression `f1()`. We have provided a template for the inverted tree, leaving blanks in lemmas (i), (iii), (v), (vi) and (vii) for you to fill out.

$b_{f1} = \texttt{y = 2; return f2()}$
$S(E(f1)) = (f2 = \texttt{def f2()} \rightarrow \texttt{int}, b_{f1}, E_{f1})$
$l_{f2} = newloc(S)$
$E' = E_{f1}[l_{f2}/f2]$
$b_{f2} = \texttt{return x + 1}$
$v_{f2} = (b_{f2}, E_{f2})$
$G, E', S \vdash \texttt{def f2()\{...\}} : v_{f2}, S, \_$ from lemma (i)
$S_1 = S[v_{f2}/l_{f2}]$
$G, E', S_1 \vdash b_{f1} : \_, S_2, int(1)$ from lemma (ii)

$G, E, S \vdash \texttt{f1()} : int(1), S_2, \_$

(i)

$b_{f2} =$
$E_{f2} =$
$v_{f2} =$

$G, E, S \vdash \texttt{def f2()\{...\}} : v_{f2}, S, \_$

(ii)

$G, E', S_1 \vdash \texttt{y = 2} : \_, S_2, \_$ from lemma (iii)
$G, E', S_2 \vdash \texttt{return f2()} : \_, S_2, int(1)$ from lemma (iv)

$G, E', S_1 \vdash b_{f1} : \_, S_2, int(1)$

(iii)

$G, E', S_1 \vdash \texttt{2} : int(2), S_1, \_$
$E'(y) =$
$S_2 = S_1[int(2)/l_y]$

$G, E', S_1 \vdash \texttt{y = 2} : \_, S_2, \_$

(iv)

$G, E', S_2 \vdash \texttt{f2()} : int(1), S_2, \_$ from lemma (v)

$G, E', S_2 \vdash \texttt{return f2()} : \_, S_2, int(1)$

(v)

$b_{f2} =$
$S_2(E'(f_2)) =$
$G, E', S_2 \vdash b_{f2} : \_, S_2, int(1)$ from lemma (vi)

$G, E', S_2 \vdash \texttt{f2()} : int(1), S_2, \_$

(vi)

$G, E', S_2 \vdash \texttt{x} : int(0), S_2, \_$ from lemma (vii)

$G, E', S_2 \vdash \texttt{return x + 1} : \_, S_2, int(1)$

(vii)

$E'(x) =$
$S_2(l_x) =$

$G, E', S_2 \vdash x : int(0), S_2, \_$

3. Suppose we introduce the `switch` expression into ChocoPy:

$$\texttt{switch}\ e : b_1\ \texttt{case}\ e_1,\ b_2\ \texttt{case}\ e_2,\ \ldots,\ b_n\ \texttt{case}\ e_n,\ b_{n+1}\ \texttt{default}$$

where e and all of the e_i and b_i are expressions, and $n \geq 1$.

We want the value of the entire `switch` expression to be the evaluation of the expression associated with the case which matches the value of the expression just after the `switch` keyword. In the following example:

```
x : str = "test2"
switch x: 1 case "test", 2 case "test2", 3 default
```

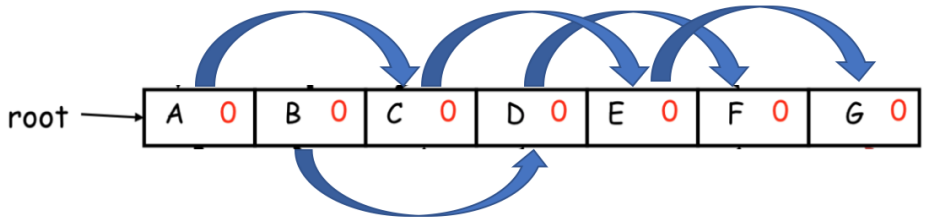
the value of `x` is "test2", which matches with the second case, and thus the whole `switch` expression evaluates to 2. In the scenario that none of the cases match then the whole `switch` expression evaluates to the body of the default case. The `switch` expression can have 1 or more cases (since $n \geq 1$), but must have a default case.

To be more formal, this hypothetical `switch` expression is evaluated in the following way: we want to determine if `e == e1`, and if so, then the result of the `switch` expression is the value of b_1. If not, then if `e == e2`, the result of the `switch` expression is the value of b_2. Evaluation proceeds in this manner, checking e against each e_i using the `==` operator. If none of the cases match, then the result of the `switch` expression is the value of b_{n+1}. We require the following properties to hold:

- If the i^{th} case of the `switch` expression matches, and no previous case matches, then the expressions $e, e_1, \ldots, e_i, b_i$ are evaluated in this order, each exactly once, and the value of the `switch` expression is the value of b_i.
- If none of the cases match, then the expressions $e, e_1, \ldots, e_n, b_{n+1}$ are evaluated in this order, each exactly once, and the value of the `switch` expression is the value of b_{n+1}.

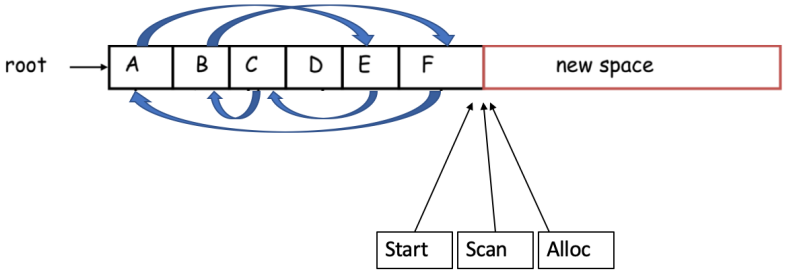
Define operational semantics rule(s) for evaluating any `switch` expression.

4. (a) This question is about the mark-and-sweep garbage collection strategy. The following figure shows the state of the heap before garbage collection, where the red zeros indicate that the mark bits are reset on each object:



Recall that the mark-and-sweep strategy is to perform the mark operation, and then the sweep operation. Show the state of the heap after each operation. Explicitly write the mark bits.

(b) This question is about the stop-and-copy garbage collector. The following figure shows the state of the heap before garbage collection:



Show the state of the heap immediately after the first two objects have been both copied and scanned.

(c) Write a ChocoPy program such that reference counting would not free an object on the heap which should be freed.

(d) Consider the following pseudocode of a program which allocates a large number of fixed-size objects, given `total_bytes` and `object_size` as inputs:

```
cumulative := 0
while cumulative < total_bytes:
    obj := allocate_object_of_size(object_size)
    # this obj is never used
    cumulative += object_size
```

i. Suppose `total_bytes` is much larger than the available memory size and `object_size` is small enough that many objects of this size can fit in memory. The program therefore allocates a large number of small objects. Between the mark-and-sweep and stop-and-copy algorithms, which garbage collection strategy would be more efficient, and why?

ii. Now suppose `total_bytes` is much larger than the available memory size and `object_size` is large enough that only a few objects of this size can fit in memory. The program therefore allocates a small number of large objects. Between the mark-and-sweep and stop-and-copy algorithms, which garbage collection strategy would be more efficient, and why?