| CS164 Programming Languages and Compilers | Spring 2024 |
|---|---|

# Written Assignment 5 Solutions

**Assigned:** March 20 **Due:** April 10 at 3:00pm

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.

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

(i)

$$\frac{\begin{array}{l}\text{x declared to be global in \_\_init\_\_}\\ E_{\_\_init\_\_} = G\\ v_{\_\_init\_\_} = (\texttt{self}, \texttt{x} = \texttt{x} + \texttt{1}, E_{\_\_init\_\_})\end{array}}{G, G, S \vdash \texttt{def \_\_init\_\_(self : "A")\{...\}} : v_{\_\_init\_\_}, S, \_}$$

(ii)

$$\frac{\begin{array}{l}bar = \texttt{def bar()} \rightarrow \texttt{int} \text{ is a nested function defined in foo}\\ E_{foo} = G\\ v_{foo} = (self, x, \texttt{bar} = bar, \texttt{return bar()}, E_{foo})\end{array}}{G, G, S \vdash \texttt{def foo(self : "A", x : int)} \rightarrow \texttt{int\{...\}} : v_{foo}, S, \_}$$

(iii)

$$\frac{\dfrac{\dfrac{\begin{array}{l}E'(x) = l_x\\ S_2(l_x) = int(1)\end{array}}{\begin{array}{l}G, E', S_2 \vdash x : int(1), S_2, \_\\ G, E', S_2 \vdash 1 : int(1), S_2, \_\\ 1 + 1 = 2\end{array}}}{\begin{array}{l}G, E', S_2 \vdash \texttt{x + 1} : int(2), S_2, \_\\ E'(x) = l_x\\ S_3 = S_2[int(2)/l_x]\end{array}}}{G, E', S_2 \vdash \texttt{x = x + 1} : \_, S_3, \_}$$

(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.

$$\frac{\begin{array}{l}v_{bar} = A(a1 = l_{a1}, a2 = l_{a2}, \_\_init\_\_ = l_{\_\_init\_\_}, foo = l_{foo})\\ G, E, S \vdash \texttt{bar} : v_{bar}, S, \_ \qquad \text{from lemma (i)}\\ S(l_{a1}) = int(0)\end{array}}{G, E, S \vdash \texttt{bar.a1} : int(0), S, \_}$$

(i)

$$\frac{\begin{array}{l} E(bar) = l_{bar} \\ S(l_{bar}) = v_{bar} \\ v_{bar} = A(a1 = l_{a1}, a2 = l_{a2}, \_\_init\_\_ = l_{\_\_init\_\_}, foo = l_{foo}) \end{array}}{G, E, S \vdash \texttt{bar} : v_{bar}, S, \_}$$

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.

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

(i)

$$\dfrac{\begin{array}{l} b_{f2} = \texttt{return x} + 1 \\ E_{f2} = E \\ v_{f2} = (b_{f2}, E_{f2}) \end{array}}{G, E, S \vdash \texttt{def f2}()\{...\} : v_{f2}, S, \_}$$

(ii)

$$\dfrac{\begin{array}{l} G, E', S_1 \vdash \texttt{y} = 2 : \_, S_2, \_ \qquad \text{from lemma (iii)} \\ G, E', S_2 \vdash \texttt{return f2}() : \_, S_2, int(1) \qquad \text{from lemma (iv)} \end{array}}{G, E', S_1 \vdash b_{f1} : \_, S_2, int(1)}$$

(iii)

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

(iv)

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

(v)

$$\frac{\begin{array}{l} b_{f2} = \texttt{return x} + \texttt{1} \\ S_2(E'(f_2)) = (b_{f2}, E_{f2}) \\ G, E_{f2}, S_2 \vdash b_{f2} : \_, S_2, int(1) \qquad \text{from lemma (vi)} \end{array}}{G, E', S_2 \vdash \texttt{f2}() : int(1), S_2, \_}$$

(vi)

$$\frac{\begin{array}{l} G, E', S_2 \vdash \texttt{x} : int(0), S_2, \_ \qquad \text{from lemma (vii)} \\ G, E', S_2, \vdash 1 : int(1), S_2, \_ \\ G, E', S_2, \vdash \texttt{x} + \texttt{1} : int(1), S_2, \_ \end{array}}{G, E', S_2 \vdash \texttt{return x} + \texttt{1} : \_, S_2, int(1)}$$

(vii)

$$\frac{\begin{array}{l} E'(x) = l_x \\ S_2(l_x) = int(0) \end{array}}{G, E', S_2 \vdash x : int(0), S_2, \_}$$

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:

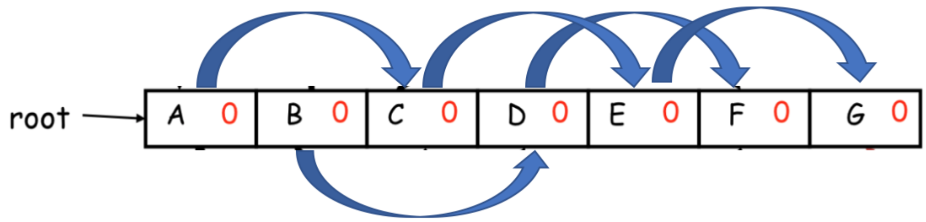
- If the $i^{\text{th}}$ case of the `switch` expression matches, and no previous case matches, then the expressions $e, e_1, \ldots, e_i, b_i$ are evaluated in this order, each exactly once, and the value of the `switch` expression is the value of $b_i$.
- If none of the cases match, then the expressions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

$$\frac{\begin{array}{l} G, E, S \vdash e : v, S_1, \_ \\ \text{for some i where } 1 \leq i \leq n \\ G, E, S_1 \vdash e_1 : v_1, S_2, \_ \\ v_1 \neq v \\ \ldots \\ G, E, S_i \vdash e_i : v_i, S_{i+1}, \_ \\ v_i = v \\ G, E, S_{i+1} \vdash b_i : v_{b_i}, S_{i+2}, \_ \end{array}}{G, E, S \vdash \texttt{switch e : b}_1 \texttt{ case e}_1, \texttt{b}_2 \texttt{ case e}_2, \ldots, \texttt{b}_n \texttt{ case e}_n, \texttt{b}_{n+1} \texttt{ default} : \ v_{b_i}, S_{i+2}, \_}$$

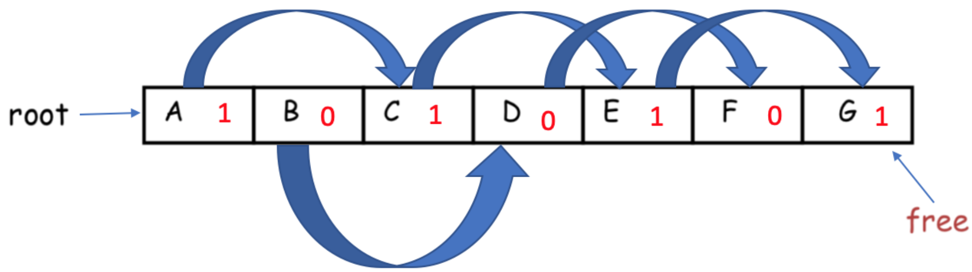
$$\frac{\begin{array}{l} G, E, S \vdash e : v, S_1, \_ \\ G, E, S_1 \vdash e_1 : v_1, S_2, \_ \\ v_1 \neq v \\ \ldots \\ G, E, S_n \vdash e_n : v_n, S_{n+1}, \_ \\ v_n \neq v \\ G, E, S_{n+1} \vdash b_{n+1} : v_{b_{n+1}}, S_{n+2}, \_ \end{array}}{G, E, S \vdash \texttt{switch e : b}_1 \texttt{ case e}_1, \texttt{b}_2 \texttt{ case e}_2, \ldots, \texttt{b}_n \texttt{ case e}_n, \texttt{b}_{n+1} \texttt{ default} : \ v_{b_{n+1}}, S_{n+2}, \_}$$

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:

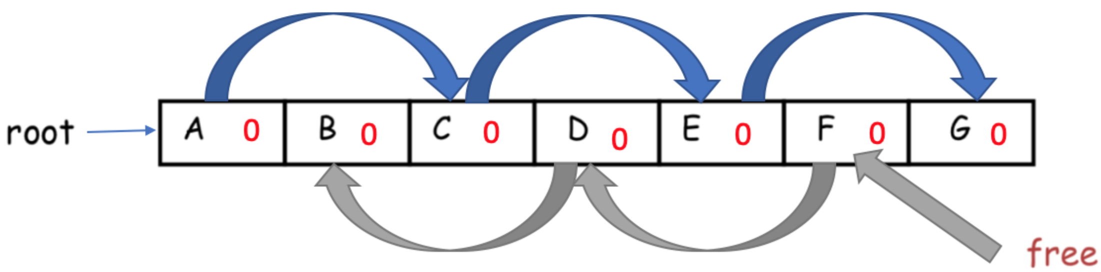



Recall that the mark-and-sweep strategy is to perform the mark operation, and then the sweep operation. Show the state of the heap after each operation. Explicitly write the mark bits.

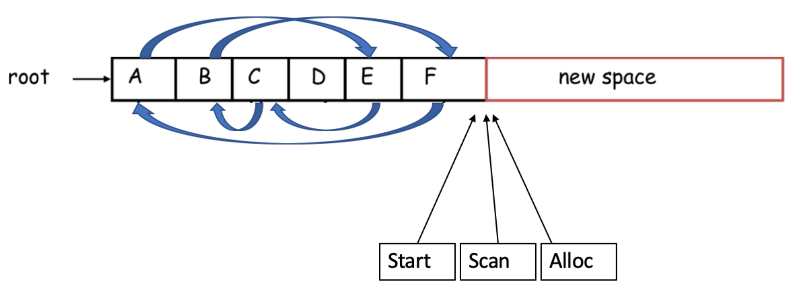
After mark:



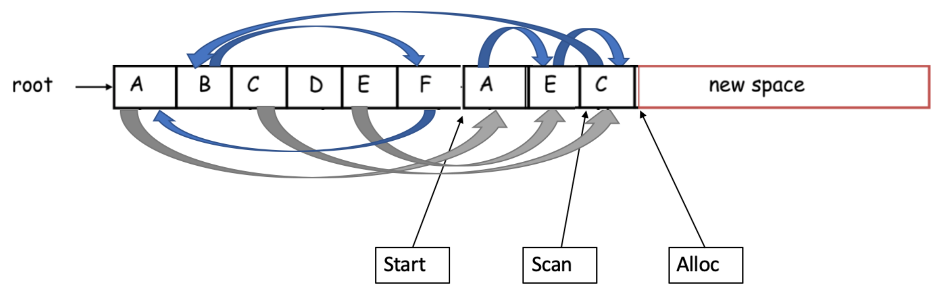

After sweep:




(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.




Answer: A will be copied thus there will be a forwarding node to the new location of A, and the alloc pointer will move forward. Then we will scan A, and thus will alloc space for E. We will add a forwarding pointer from the old E to the new E. We are done scanning A so the scan pointer can be moved forward. We then scan E and thus add C to the new space and move the alloc pointer forward, and we add a forwarding pointer from old C to new C. We then move scan pointer forward since we are done scanning E. Thus we have reached the point where the first two objects are copied and scanned.

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

Answer: Student answers will vary but we are looking for a cyclical structure of unreachable objects which will prevent reference counting from correctly garbage collecting. One simple example is illustrated here with a Node class that represents nodes in a linked list that contains integers with a first and next attribute.

```
class Node(object):
   first: int = 1
   next: Node = None
x: Node = None
x = Node()
x.next = x
x = Node()
```

(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?

Answer: stop-and-copy, since it will only copy at most one reachable object, whereas mark-and-sweep will have to iterate through almost all small objects which were allocated in order to add them to the free list.

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?

Answer: mark-and-sweep, since it will only have to mark and sweep a few large objects, while stop-and-copy may have to actually copy a large object (maybe several GB!) to the new-space each time memory fills up due to the allocations.