Note: The assignment is written in landscape mode to make reading of Figure 1 easier on a computer screen.

The program in questions produces varying output depending on the order of execution. There are termination and non-termination scenarios as well as scenarios in which the order of execution affects the final values of `x` and `y`. Let us label the thread containing the `while` loop *Thread-A* and let us label the thread containing `<await (x==y)>` *Thread-B.* Initially, there is only possible interleaving of the threads - *Thread-A* executes 10 iterations decrementing the value of `x` by one each iteration. When the value of `x` is decremented to 0, a first 'interesting' interleaving of the two threads can begin. The scenarios discussed below will assume this stage as the starting point. *Figure 1* outlines the tree of executions, labels such as **A** and **B** are used to refer to individual subtrees and simplify explanation.

Firstly, an interesting interleaving happens at **A**, where the program can decide to execute *Thread-A* and exit the loop or *Thread-B* and unblock. Let us consider the scenario where the while loop is exited (**B)**. At **B**, if we choose to increment the value of y, the program will not be able to terminate as the values of x and y are (0, 1) respectively (**C)**. Alternatively, executing the await and unblock *Thread-B* leads to termination scenarios.

Secondly, starting **D**, the execution can proceed by setting the value of x without any impact on the final state. At **E**, after the value of x is set to 8, we need to sequentially set values of y. This can lead to four different outcomes. Executing *y = 2* first will result in final values (8, 3). Choosing to execute *y = y + 1* first, we are presented with a race condition issue where the above statement may require multiple instructions to execute - exposing itself to a race condition. Depending on the order of execution, shown as the subtrees of **F**, we can end up with the following final values: (8, 1), (8, 2), (8, 3).

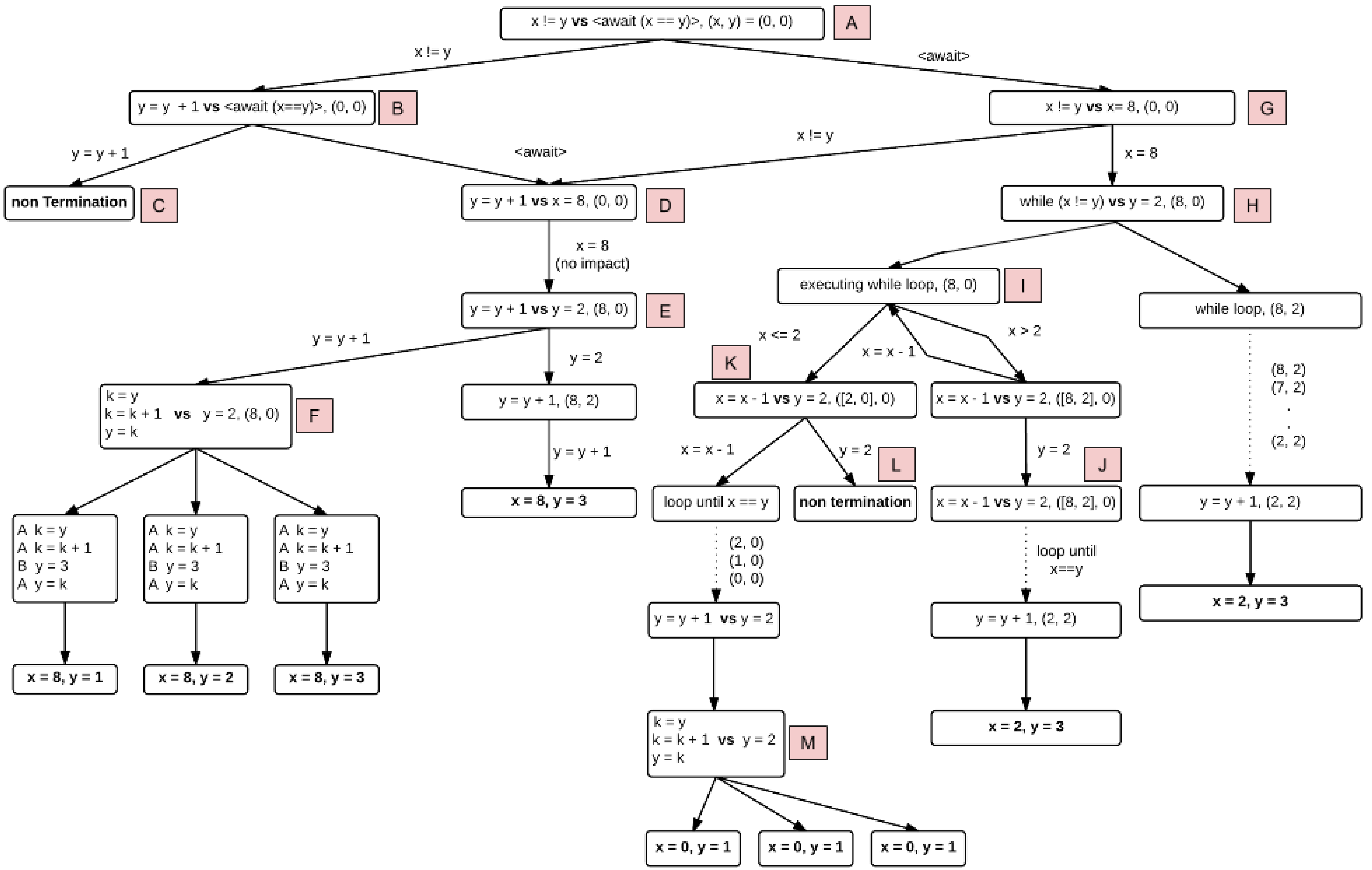
Thirdly, following the execution of the *await* form the root (**A)**, we effectively unblock *Thread-B* execution. At **G**, we can either execute the condition check of the *while* loop or set the value of x to 8. Deciding to execute the while loop, we arrive at a previously discussed tree **D**. Choosing to execute *x = 8* leaves us at subtree **H**.

Starting at **H**, if we choose to execute *y = 2*, we can proceed with execution of the *while* loop in *Thread-A* until x and are (2, 2) respectively. At this point, the only instruction to execute is to increment y, arriving at final values (2, 3). Choosing to execute the condition check of the *while* loop first (at **H**), we reach a slightly more complicated execution pattern. We can either keep decrementing *x* as part of the *while* loop, or at any time we may set the value of *y* to 2. If setting the value of *y* to 2 is executed before *x is* 2 or less, we reach **J**. At this stage, we keep decrementing the value of *x* until it is equal to 2. At this stage, we only increment the value of *y* and arrive at (2, 3).

Choosing to set the value of *y* to 2 when the value of *x* is 2 or less inside the loop, we can arrive at two scenarios. Either we are still executing the *while* loop, in which case the program never terminates (**L**) or the value of *y* is set only after we have exited the *while* loop, in which case we reach **M**. At this point, depending on the precise order of execution of *y = y + 1* and *y = 2*, we can arrive at final values (0, 1), (0, 2), (0, 3).

To conclude, there are 2 non-termination scenarios, and a 9 interesting sequentially consistent outcomes of execution. The main reason for the large number of outcomes is due to race conditions arising from non-atomic operations such as *x != y* and *y = y + 1*. Furthermore, deadlock and a while-loop run off are another reasons for the large number of outcomes.

Figure 1


x != y **vs** <await (x == y)>, (x, y) = (0, 0)
A
x != y
<await>
y = y + 1 **vs** <await (x==y)>, (0, 0)
B
x != y **vs** x= 8, (0, 0)
G
y = y + 1
<await>
x != y
x = 8
non Termination
C
y = y + 1 **vs** x = 8, (0, 0)
D
while (x != y) **vs** y = 2, (8, 0)
H
x = 8
(no impact)
executing while loop, (8, 0)
I
y = y + 1 **vs** y = 2, (8, 0)
E
while loop, (8, 2)
y = y + 1
y = 2
x <= 2
x = x - 1
x > 2
K
k = y
k = k + 1 **vs** y = 2, (8, 0)
y = k
F
y = y + 1, (8, 2)
x = x - 1 **vs** y = 2, ([2, 0], 0)
x = x - 1 **vs** y = 2, ([8, 2], 0)
(8, 2)
(7, 2)
.
.
(2, 2)
y = y + 1
x = x - 1
y = 2
L
y = 2
J
x = 8, y = 3
loop until x == y
non termination
x = x - 1 **vs** y = 2, ([8, 2], 0)
y = y + 1, (2, 2)
A k = y
A k = k + 1
B y = 3
A y = k
A k = y
A k = k + 1
B y = 3
A y = k
A k = y
A k = k + 1
B y = 3
A y = k
(2, 0)
(1, 0)
(0, 0)
loop until
x==y
x = 2, y = 3
y = y + 1 **vs** y = 2
y = y + 1, (2, 2)
x = 8, y = 1
x = 8, y = 2
x = 8, y = 3
x = 2, y = 3
k = y
k = k + 1 **vs** y = 2
y = k
M
x = 0, y = 1
x = 0, y = 1
x = 0, y = 1