Explanation and Proof of Correctness of Our Approach to Part 1

Proof of Correctness of our Approach

In our approach to the problem, we create two refs which will traverse the given mutable list. These refs are denoted `fast` and `slow` in our code. In each iteration, `fast` travels two nodes down the mutable list and `slow` travels one node down the mutable list. We make the following claim:

Claim. The `fast` and `slow` refs become physically equal at any time after the start if and only if the mutable list contains a cycle.

Proof.
($\Rightarrow$): We must prove that if `fast == slow` at some time after the start, the mutable list contains a cycle. This is equivalent to proving that if the list in fact contains no cycle, then `fast` never physically equals `slow` at any time after the start. Indeed this is obvious because if there is no cycle, then `fast` surely arrives at the end of the list (i.e. it becomes `Nil`) before `slow`, causing the termination of `cycle_len_start_helper` before `slow` physically equals `fast`.

($\Leftarrow$): We must prove that if the mutable list contains a cycle, then `fast == slow` at some time after the start. Consider any arbitrary list with a cycle of arbitrary length starting at an arbitrary node. The key insight here is that once either `fast` or `slow` enters the cycle, it never leaves the cycle by the very definition of a cycle (each node is the `Cons` of a value and the next node in the cycle, so when we advance our refs they stay within the cycle).

Thus, when `slow` enters the cycle, `fast` is surely still in the cycle. Let the distance of `slow` from `fast` at this time be n (i.e. `slow` is n nodes in front of `fast` in the direction of `fast`'s travel along the cycle). However, since in each iteration `fast` advances two nodes along the cycle whereas `slow` advances one node, the distance between the two decreases by 1 node for each iteration. Thus, as long as the cycle is not of infinite length, at some time after the start `fast` must equal `slow` when the distance n between them decreases to 0.

Application to problem requirements

Besides a Boolean indicating the presence or absence of a cycle, `cycle_len_start_helper` also returns the length of the cycle (if it exists and -1 otherwise), the length of the list prior to the start of the cycle (which equals the overall length of the list if no cycle is present), a reference to the node at which the cycle starts, and a reference to the predecessor of that node in the cycle. The remainder of this document explains the calculation of these return values.

Cycle Length. If a cycle does not exist, we return -1 (to distinguish it from any valid cycle length). If a cycle exists, then from our approach to ascertain the existence of the cycle, we know that the node at which `fast == slow` is surely a node on the cycle. Then, we simply advance a new ref from that node, incrementing a counter by 1 each time starting from 0. When the new reference is physically equal to `fast` (i.e. has returned to the node at which it started) then the value of the counter surely equals the length of the cycle.

Cycle Start Node. In order to calculate the cycle start node (to visualize, this is the first node with value 2 in the diagram given in the problem set 6 spec), we create two references, `ahead` and `behind`. We advance `ahead` by a distance equal to the cycle length we just calculated while leaving `behind` at the start of the list. Then, we advance the two references in tandem (using `tandem_advance_to_start`). When these two references become equal, the node at which they become equal is surely the head of the cycle.

Length Prior to Cycle. If there is no cycle, this is trivially calculated by incrementing a counter every time `fast` advances. If there is a cycle, this is calculated with a counter that increments in tandem with the `behind` reference's advance to the start of the cycle (since `behind` travels from the start of the list to the start of the cycle, thereby yielding the length we desire).

Predecessor to Cycle Start in Cycle. In `tandem_advance_to_start`, as `ahead` is being advanced, we keep track of the predecessor to `ahead` as a reference. Upon termination of this function, we return this predecessor, which now must refer to the node in the cycle prior to the start of the cycle.

We already described how to determine whether a cycle exists. The `flatten` function is easily written by setting the predecessor to the cycle start to a mutable list with value equal to the `Cons` its original value and `Nil`, which effectively removes its connection to the start of the cycle and thereby "flattens" the list. The `mlength` function trivially returns the sum of the length prior to the cycle and the pre-cycle length if a cycle exists, or only the pre-cycle length if a cycle does not exist.