Code Review 6: March 27, 2020

Tail Recursion, Imperative Programming, and Graphs

Hakeem Angulu, hangulu@college.harvard.edu

Learning Goals

- Understand the difference between tail recursive functions and their non-tail-recursive counterparts
- Be able to convert basic recursive functions into their tail-recursive counterparts
- Understand structural vs. physical equality
- Understand the use cases of refs
- Be introduced to graphs and basic graph searching with BFS and DFS

Tail Recursion

Recursion is incredibly powerful, and we've been using it quite effectively on many problems thus far. However, for large inputs, some functional recursive implementations may run into stack overflow errors. Let's look at the following example:

```
let rec sum (lst : int list) : int =
  match lst with
  | [] -> 0
  | hd :: tl -> hd + sum tl ;;
```

As noted in Lab 13, each call of `length tl` is suspended until we reach the end of the list. At that point, the stack finally unwinds and the expression is evaluated. One way of addressing this problem is with **tail recursion**.

Tail recursive functions **have the recursive invocation as the result of the invoking call (that is, the final computation to find a result is the recursive call)**.

Thus, functions need not use a call stack to keep track of suspended computations, avoiding the problem of stack overflow. Let's look at the tail-recursive version of sum:

```
let sum (lst : int list) : int =
  let rec sum_tr lst acc =
    match lst with
    | [] -> acc
    | hd :: tl -> sum_tr tl (hd + acc) in
  sum_tr lst 0 ;;
```

The technique used here, using a tail-recursive auxiliary function that makes use of an added argument that acts as an accumulator for the partial results, is a common one for converting functions to tail-recursive form.

Remember `List.fold_left` and `List.fold_right`? Well, other than left- and right-associativity, the big difference between the two is that `List.fold_left` is tail-recursive, while `List.fold_right` is not.

One can see the difference plainly by examining the source code for these functions:

```
let rec fold_left f accu l =
  match l with
    [] -> accu
  | a::l -> fold_left f (f accu a) l

let rec fold_right f l accu =
  match l with
    [] -> accu
  | a::l -> f a (fold_right f l accu)
```

Refs

From Cornell's *Data Structures and Functional Programming* course:

> There are only two built-in mutable data structures in OCaml: refs and arrays. OCaml supports imperative programming through the primitive parameterized ref type. A value of type "int ref" is a pointer to a location in memory, where the location in memory contains an integer. Like lists, refs are polymorphic, so in fact, we can have a ref (i.e., pointer) to a value of any type.

Imperative programming languages have many more mutable structures, but OCaml's focus on functional programming allows it to limit the prevalence of these data structures. Instead, it's encouraged for you to think functionally, and to avoid side effects.

However, side effects are sometimes useful. Printing to the console is a side effect, for example.

One can think of a ref as **a box in which some value is stored and able to be replaced**. By using the `:=` operator, the value in the box can be changed as a side effect.

It is important to distinguish between the value that is stored in the box, and the box itself (the ref).

Physical vs. Structural Equality

Structural equality is the `=` operator in OCaml, and asks whether two things have the same value - aka are my two operands ”structurally” the same.

Physical equality is the `==` operator in OCaml, and asks whether two things are stored in the same location in memory - aka are the two operands ”physically” equal.

Exercise 1:

```
let a = ref 1;;
let b = ref 1;;
let c = ref a;;
```

Which of the following evaluate to true? 1. `a = b` 2. `a == b` 3. `!a = !b` 4. `!a == !b` 5. `c = a` 6. `!c = a` 7. `!c==a` 8. `!c = b` 9. `!c==b` 10. `!(!c) = !b` 11. `!(!c) == !b`

Remembering Remember

A tricky problem every year is the `remember` function from Lab 12. It was as follows:

Write a function `remember` that returns the last string that it was called with. The first time it is called, it should return the empty string.

```
# remember "don't forget" ;;
- : string = ""
# remember "what was that again?" ;;
- : string = "don't forget"
# remember "etaoin shrdlu" ;;
- : string = "what was that again?"
```

One solution to this function is:

```
let memory = ref "" ;;

let remember msg =
  let previous = !memory in
  memory := msg;
  previous ;;
```

This is generally fine, but the main problem is that it allows one to access, and potentially change, the contents of `memory` outside the function `remember`. That is undesirable behavior.

One might acknowledge this drawback and begin to design a new solution that takes advantage of our scoping lessons from this class:

```
let remember msg =
  let memory = "" in
  let previous = !memory in
  memory := msg;
  previous ;;
```

However, this does not work. Every time you run this function, it will return the empty string. That is because the ref `memory` is redefined every time you run the function, because its definition is within the body of the function.

We can work around this limitation with the following, final solution:

```
let remember : string -> string =
  let memory = ref "" in
  fun (msg : string) -> let previous = !memory in
                        memory := msg;
                        previous ;;
```

Exercise 2:

Why does this work?

This works because the value `remember` is defined by a let statement, and thus follows the rules for substitution and evaluation we learned a few weeks ago. By those rules, we substitute the value `memory` into the anonymous function once, then `remember` gets evaluated to be that anonymous function. Therefore, the line `let memory = ref ""` is only evaluated once, but the anonymous function gets to keep using the ref.

Mutable Lists

We were introduced to the idea of the **mutable list** this week. Here is the type definition:

```
type 'a mlist =
| Nil
| Cons of 'a * ('a mlist ref)
```

If you're comfortable with the idea of a **linked list**, that may be a good way to interpret the mutable list above. Every element of the list leads to a reference of the rest of the list.

We had a good bit of practice with them in Lab, but please reach out if you have any more questions.

Graphs

A few weeks ago, we were introduced to the idea of **trees**, which are simply **graphs without cycles**. This week, especially in the problem set, you all are working with **graphs**, and learning how to search them effectively.

Here is some pseudocode for how to search a graph: 1. pick a start vertex and a goal vertex 2. from the start vertex, add its **neighboring vertices** to a collection of **pending** vertices, sometimes called the **frontier** 3. remove a vertex from the frontier, check if it is the goal. if it is, you're done. if not, add all of its neighboring vertices that you **have not yet visited** to the frontier, and continue 4. if you exhaust your frontier and have no more vertices to visit, there is no path to the goal vertex

The difference between the two data structures we'll be using for graph search in this class, Breadth-First Search (BFS) and Depth-First Search (DFS), is the type of frontier you use: - BFS -> queue - DFS -> stack

View the accompanying videos, created by Brian Yu, to see how these frontiers result in different behavior.

Exercise 3:

What are the advantages and disadvantages of DFS and BFS? When would you want to use one over the other?

The solutions to all of the above exercises will be available on Sunday at noon.

For this and future code reviews, please do not hesitate to reach out to me with any questions or concerns. My email can be found at the top of this document.