CS51 **Spring 2024**

Code Review 6: Mutable State and Imperative Programming

1 Side Effects

Definition 1.1 (Side Effect)**.** A function has a **side effect** if it modifies some state variable outside its local scope. Examples include a function modifying a non-local variable, modifying a *mutable* argument passed into the function, and printing out a value to standard output.

A function that has **no side effects** is called a **pure** function.

We suggest you read the start of Chapter 15 in the textbook for the discussion about imperative programming and side effects.

2 References

Definition 2.1 (References)**.** A **reference** is a location ℓ of a block of mutable memory containing a value v. We say that a reference r evaluates to a location ℓ whose contents are value v (i.e. $\ell \mapsto v$).

`OCaml` provides an abstract, polymorphic type, `'a ref` to represent references. A reference has type `'a ref` if its reference refers to memory containing a value of type `'a`. Like functions, references are **first-class values**.

```
# let r : int ref = ref 42 ;;
 val r : int ref = {contents = 42}
```

Here, we create a reference to a block of memory storing the value `42`.

`OCaml` references support the following **operations**:

- `ref : 'a -> 'a ref`: **Creates** a new reference to a value of type `'a`
- `(!) : 'a ref -> 'a`: Accepts a reference and returns the referenced value (known as **dereferencing)**
- `(:=) : 'a ref -> 'a -> unit` : **Updates** the value stored at a reference

Example 2.2 (References Example)**.** Here is an example of `OCaml` references being used.

```
# let r = ref 42 ;;
val r : int ref = {contents = 42}
# let s = ref 42 ;;
val s : int ref = {contents = 42}
# r := 21 ;;
- : unit = ()
# s := !r ;;
- : unit = ()
# let s = r ;;
val s : int ref = {contents = 21}
# r := 7 ;;
- : unit = ()
```

```
# !r, !s ;;
- : int * int = (7, 7)
```

Please see section 15.1.1, Figure 15.1 for a visualization (box and arrow diagram) of what's going on here. Make sure you can follow everything that happens.

2.1 Binary Sequencing

Definition 2.3 (Binary Sequencing)**.** In `OCaml`, `P; Q` means that we evaluate expression `P` then expression `Q`. Expression `P` **must have type unit**. `P; Q` returns the value of expression Q.

Example 2.4 (Binary Sequencing Example)**.** Here is an example of binary sequencing being used.

```
# let x =
   let y = ref 10 in
   y := !y * 2; !y ;;

#  val x : int = 20
```

2.2 Other Mutable Data Types

In addition to references, `OCaml` supports other built-in mutable data types. **Please read section 15.2 of the textbook**.

2.3 Structural Equality and Physical Equality

Definition 2.5 (Structural Equality)**.** Two values are **structurally equal** if they have the same structure.

In `OCaml`, we compare structural equality using the = operator. To check for **inequality**, we use the <> operator. If `x = y` evaluates to `true`, then `x` and `y` are structurally equal).

Definition 2.6 (Physical Equality)**.** Two values are **physically equal** if they are identical physical objects in memory.

In `OCaml`, we compare physical equality using the == operator. To check for **inequality**, we use the `!=` operator. If `x == y` evaluates to `true`, then `x` and `y` are physically equal.

Proposition 2.7 (Physical Equality Implies Structural Equality)**.** Suppose `x` and `y` are two values. If `x == y`, then `x = y`.

Physical equality implies structural equality because if two values are identical objects in physical memory, then they have the same structure. Note that the **converse** (structural equality implies physical equality) is false.

Example 2.8 (Structural and Physical Equality Example)**.** Consider the following values:

```
let w = ref "CS51" ;;
let x = ref "CS51" ;;
let y = ref "CS124" ;;
let z = x ;;
```

`w` and `x` are structurally equal, but not physically equal. They're both references that refer to a memory location with value `"CS51"`, but they refer to two different memory locations.

Likewise, `w` and `y` are neither structurally equal nor physically equal. `z` and `x` are both physically equal and structurally equal.

We suggest you experiment in the `REPL` to see what types of values are structurally and physically equal in OCaml.

2.4 Example: Functions with References

Example 2.9 (Remember)**.** In lab, we defined a function `remember` that returns the last string that a function was called with. We can define this function using references.

Listing 1: Remember with External Reference

```
let memory = ref "" ;;

let remember (msg : string) : string =
 let previous = !memory in
 memory := msg;
 previous ;;
```

Here, we define a *global* reference `memory` outside the function that stores the last string `remember` was called with. However, there's a problem with this implementation: a user can access the contents of `memory` without using the function, violating the **edict of prevention**.

Suppose we define `memory` inside the body of the function:

Listing 2: Remember with Internal Reference (Incorrect)

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

The problem here is that every time the function is called, `memory` is re-initialized, so the function will not behave correctly.

Listing 3: Remember with Internal Reference (Correct)

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

Here, `memory` is defined within the **scope** of the function, so it cannot be accessed outside the function. In addition, we resolve the problem with the second approach allowing `memory` to **out scope the function definition but not the naming**.

3 Mutable Lists

In addition to references, `OCaml` provides other primitive mutable data types (**see Section 15.2**). Using references, we can construct mutable *composite* data types. **Mutable Lists** are an example of a mutable data structure.

Definition 3.1 (Mutable Lists)**.** Here's how we define a mutable list:

Listing 4: Mutable Lists

```
type 'a mlist = 'a mlist_internal ref
 and 'a mlist_internal =
   | Nil
   | Cons of 'a * 'a mlist ;;
```

The type definition is similar to lists, except that lists are either references to `Nil` or references to a pair consisting of a value and a reference to the rest of the list.

Example 3.2. Here's an example of mutable lists being defined:

```
# let r : int mlist = ref Nil ;;
val r : int mlist = {contents = Nil}
# let s : int mlist = ref (Cons (1, r)) ;;
val s : int mlist = {contents = Cons (1, {contents = Nil})}
# let t: int mlist =  ref (Cons (2, s)) ;;
val t : int mlist = {contents = Cons (2, {contents = Cons (1, {contents = Nil})})}
```

Example 3.3 (mappend)**.** In lab 12, we defined a function `mappend` that appends two mutable lists.

Listing 5: mappend

```
let rec mappend (xs : 'a mlist) (ys : 'a mlist) : unit =
  match !xs with
   | Nil -> xs := !ys
   | Cons (_hd, tl) -> mappend tl ys ;;
```

Here, we want to append `ys` to the end of `xs`. To do so, we can have the end of `xs` (which references `Nil`), reference the list `ys`. In the recursive case, calling `mappend tl ys` will allow us to traverse `xs` until we get to the end of the list.

4 Procedural Programming

Procedural programming is a programming paradigm based on the idea that programs are a series of procedures to be carried out. In other words, programs are *commands* with *side effects* having loops that execute commands repeatedly.

4.1 Loops

Syntax 4.1 (Loops)**.** In `OCaml`, we write for loops and while loops using the following syntax:

```
for <variable> from <expr_start> to <expr_end> do
   <expr_body>
done
```

```
while <expr_cond> do
   <expr_body>
done
```

Example 4.2 (Odd)**.** In lab 13, we defined functions `odd_for` and `odd_while` that return a list of odd integers less than or equal to a given int.

Listing 6: Odd

```
let odd_while (limit : int) : int list =
  let counter = ref 1 in
  let lst_ref = ref [] in
  while !counter <= limit do
    lst_ref := !counter :: !lst_ref;
    counter := !counter + 2
  done;
  List.rev !lst_ref ;;

let odd_for (limit : int) : int list =
  let lst_ref = ref [] in
  for i = 1 to (limit + 1) / 2 do
    lst_ref := (i - 1) * 2 + 1 :: !lst_ref
  done;
  List.rev (!lst_ref) ;;
```

Review the solutions for Exercise 5 of Lab 13 for discussion on the various approaches to writing this function.

4.2 Recursion vs Iteration

A procedural, iterative approach is generally more *space efficient* than a recursive, functional approach.

Definition 4.3 (Stack Frame)**.** A **call stack** is a **stack** that stores information about each subroutine or computation (e.g. function applications, operator applications) of the program. Each element in the stack is known as a **stack frame**. When we **suspend** a computation, we **push** a stack frame for that computation on to the call stack. When we **pop** a stack frame from the call stack, that means we have performed the computation.

Definition 4.4 (Stack Overflow)**.** A **stack overflow error** occurs when a program tries to push more space than allocated for the call stack.

Example 4.5 (Length)**.** Suppose we wanted to find the length of the list.

We could define a **recursive** function, with calls that look like this:

In this example, at a high level, recursive calls and additions are pushed and popped from the stack in an order such that we execute the function. For simplicity, we can just imagine that only contains recursive calls or calls to user-defined helper functions create additional stack frames.

First, `length [1; 2; 3]` would be pushed on to the stack, then `length [2; 3]`, and so on until we get to our base case.

Our stack grows to be `[length [1; 2; 3]; length [2; 3]; length [3]; length []]`.

We then pop `length []` from the stack, which evaluates to 0, then pop `length [3]` and evaluate $1+0$, pop `length [2; 3]` and evaluate $1+(1+0)$, and so on.

Since we need a stack frame for each recursive call and corresponding additions, the number of stack frames needed here is $O(n)$, where n is the length of the list. In other words, the number of stack frames grows linearly in the length of the list.

Now consider an **iterative** approach:

Listing 7: length_iter

```
let length_iter (lst : 'a list) : int =
  let counter = ref 0 in (* initialize the counter *)
  let lst_ref = ref lst in (* initialize the list *)
  while !lst_ref <> [] do (* while list not empty... *)
    incr counter; (* increment the counter *)
    lst_ref := List.tl !lst_ref (* drop element from list *)
  done;
  !counter ;;
```

Here, `length_iter` uses no (or at least fewer) stack frames (than the recursive approach), as we don't need to suspend any function calls.

When we compute a very long list, for example, a recursive approach has the risk of pushing too many stack frames to the **call stack**, resulting in a **stack overflow**.

```
# let very_long_list = List.init 1_000_000 (fun x -> x) ;;
val very_long_list : int list = [0; 1; 2; 3; 4; 5; 6; 7; ...]

# length_iter very_long_list ;;
- : int = 1000000

# length very_long_list ;;
Stack overflow during evaluation (looping recursion?).
```

For instance, as shown in the textbook, the iterative version of `length` is able to compute the length of longer lists than the recursive version.

Definition 4.6 (Tail Recursion)**.** A **tail-recursive** function is a function in which the last computation executed by a function is a recursive call.

Example 4.7 (Example 4.5 Continued)**.** Consider the following two implementations of the length function.

Listing 8: length

```
let length_tr lst =
  let rec length_plus lst acc =
    match lst with
    | [] -> acc
    | _ :: tl -> length_plus tl (1 + acc) in
  length_plus lst 0 ;;
```

```
7
8  let rec length lst =
9    match lst with
10   | [] -> 0
11   | _ :: tl -> 1 + length tl ;;
```

`length_tr` is tail recursive, as in the recursive step, the last computation in `length_plus` is a recursive call to `length_plus`. In contrast, `length` is not tail recursive because the last computation done is an addition, after we compute `length tl`.

5 Computational Efficiency

In computer science, we often analyze how fast algorithms are. To do this, we need to be able to abstractly characterize how efficient algorithms are.

Definition 5.1 (Time Complexity)**. Time Complexity** describes the amount of time it takes to run an algorithm.

1. The time complexity can be thought of a function that takes in the size of the input and returns the number of basic steps needed to run the algorithm on that input.
2. For example, suppose we define a sorting algorithm that sorts a list. We can define $T_{sort}(n)$ to be the runtime of the sorting algorithm, where n is the length of the step.
3. If $T_{sort}(n) = n \log n$, that means, for a list of length n, the sorting algorithm takes exactly $n \log n$ steps to sort the list.

When analyzing an algorithm's runtime, we have several options: we could consider the time required to run the algorithm on **average**, or the time required to run the algorithm in the **worst case**. In this class, we will just focus on analysis of worst-case time complexity.

5.1 Big-O Notation

We usually focus runtime analysis on **large input sizes**, as input sizes are often large in practice and runtime because more significant on larger input sizes. The runtime of an algorithm as the input size grows large is known as the **asymptotic runtime**.

Example 5.2 (Sum)**.** Consider the tail-recursive `sum_tr` function we wrote in lab 13.

Listing 9: sum_tr

```
1  let sum (lst : int list) : int =
2    let rec sum_tr lst accum =
3      match lst with
4      | [] -> accum
5      | hd :: tl -> sum_tr tl (hd + accum) in
6    sum_tr lst 0 ;;
```

$T_{sum}(n) = c \cdot n + k$, where n is the length of a list, and c and k are some constants. If we treat arithmetic operations and returning from the function as basic steps, then we traverse each of the n elements of the list, add that element to the accumulator, and then return the accumulator.

We notice how it gets cumbersome to define what truly counts as one basic step and define the exact theoretical runtime. For instance, while we know addition takes a *constant* number of steps, it's arbitrary whether it takes exactly 1 step, or 2-3 steps (i.e if we count reading the values being added), etc.

Also, since we're **concerned with asymptotic runtime**, these details don't matter for large values of n, where $c \cdot n + k \approx n$ since constant factors won't matter that much.

To characterize asymptotic runtime, we borrow from mathematics **Big-O Notation**, which expresses a function's *growth rate*, the behavior of a function as it approaches infinity.

Definition 5.3 (Big-O)**.** Suppose we have a function f. $O(f)$ is the **set** of functions where $g \in O(f)$ if there exists constants c, n_0, such that $g(n) \leq c \cdot \dot{f}(n)$ for $n > n_0$,

Informally, we say $g \in O(f)$ if g grows at most as quickly as f.

Another definition uses limits:

If $f \in O(g)$, then $\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$.

For example, $5n^2 \in O(n^3)$ and $3n^2 + 5n + 6 \in O(n^2)$.

Now, that we have defined Big-O, we can say that $T_{sum}(n) \in O(n)$.

Theorem 5.4 (Properties of Big-O)**.** Here are some properties of Big-O. Suppose we have functions f and g where $g \in O(f)$. Then:

1) $f \in O(f)$
2) $g + k \in O(f)$ where k is a constant.
3) If $k \cdot g \in O(f)$ where k is a constant
4) If $f \in O(n^k)$ and $g \in O(n^c)$ where $k > c$, then $f + g \in O(n^k)$
5) If $f' \in O(f)$ and $g' \in O(g)$, then $f' + g' \in O(f + g)$ and $f' \cdot g' \in O(f' \cdot g')$.

Proposition 5.5 (Complexity Classes)**.** Here are commonly-used Big-O complexity classes, ranked from classes of functions that grow the slowest to fastest.

1) Constant: $O(1)$
2) Logarithmic : $O(\log n)$
3) Linear : $O(n)$
4) LogLinear : $O(n \log n)$
5) Quadratic : $O(n^2)$
6) Cubic : $O(n^3)$
7) Exponential : $O(2^n)$

5.2 Recurrences

Definition 5.6 (Recurrence Equation)**.** A recurrence equation is an equation defined in terms of itself.

For instance, $T(n) = T(n-1) + c$ is a recurrence equation since $T(n)$ is defined in terms of $T(n-1)$.

Just like with recursive functions, we can define **base cases**, such as $T(0) = c$.

Unfolding is a technique to solve for the **closed-form** function solution to the recurrence relation.

Example 5.7 (Length)**.** Recurrences are helpful in analyzing the runtime of recursive functions whose runtimes on an input depend on the runtime of the recursive function on smaller inputs. For example, consider a recursive implementation of `length`, which takes the length of the list.

Listing 10: length

```
let rec length (lst : 'a list) : int =
  match lst with
  | [] -> 0
  | _ :: t -> 1 + length t
```

We assume that arithmetic and comparison operations and reading and returning values are all $O(1)$ (constant-time operations).

$T_{length}(0) = c$ and $T_{length}(n) = T_{length}(n-1) + c$ for $n \geq 1$, where c is a constant.

To unfold, we start writing out $T_{length}(n)$ in terms of other terms of the sequence.

1) $T_{\texttt{length}}(n-1) = T_{length}(n-2) + c$
2) $T_{\texttt{length}}(n-2) = T_{length}(n-3) + c$
3) By substitution, $T_{\texttt{length}}(n-1) = T_{length}(n-3) + 2c$
4) By substitution, $T_{\texttt{length}}(n) = T_{length}(n-3) + 3c$.
5) We notice the pattern that $T_{length}(n) = T_{length}(n-k) + kc$ for some constant k.
6) At some point, we will reach our base case at 0, where $n - k = 0$, which means $n = k$.
7) We can substitute n for k above to get a **closed-form** solution for T_{length}: $T_{length}(0) + nc = c + nc \in O(n)$.

Thus, we have shown that $T_{length}(n) \in O(n)$.

Proposition 5.8 (Common Recurrence Patterns)**.** Many recursive algorithms follow common recurrence patterns, so we can often directly write the closed form solution rather than rederiving the closed-form.

Table 14.1: Some common recurrence patterns and their closed-form solution in terms of big-O.

$T(n) = c + T(n-1)$	$T(n) \in O(n)$
$T(n) = c + k \cdot n + T(n-1)$	$T(n) \in O(n^2)$
$T(n) = c + k \cdot n^d + T(n-1)$	$T(n) \in O(n^{d+1})$
$T(n) = c + 2 \cdot T(n/2)$	$T(n) \in O(n)$
$T(n) = c + T(n/2)$	$T(n) \in O(\log n)$
$T(n) = c + k \cdot n + 2 \cdot T(n/2)$	$T(n) \in O(n \cdot \log n)$

Figure 1: Common Recurrence Patterns (Table 14.1 in the textbook)

6 Practice Problems

Problem 6.1 (Generators)**.**

1. Define a function `mem_factorial` which takes a unit as an argument and returns 1 the first time it is called, returns 2 the second time it is called, returns 6 the third time it is called, and so on. In general, the function should return n! on the nth time it is called.
2. Define a function that generates the Fibonacci numbers the ith time the function is called.

   Recall that $F_1 = 1$, $F_2 = 1$, and $F_i = F_{i-1} + F_{i-2}$ for all $i > 2$.

Problem 6.2 (Tail Recursion)**.**

1. Define a non-recursive implementation of `List.filter`
2. `range n m` returns a list that contains all the integers from `n` to `m-1` inclusive. Define three versions of the `range` function: (1) a tail-recursive version, (2) a non-tail recursive version, and (3) a procedural implementation (using loops).
3. Discuss the advantages and disadvantages between a tail-recursive, non-tail-recursive, and iterative approaches.

Problem 6.3 (Mutable List Practice)**.**

1. Define a function that merges two sorted (in ascending order) mutable lists. This function should return a value of type unit and the first list should become the merging of the two lists.

Problem 6.4 (Big-O)**.** State which complexity classes (listed in proposition 5.5) that the following functions fall under [1]:

1. $f(n) = 3n^2 + 10n \log n + 1000n + 4 \log n + 9999$
2. $f(n) = (\log n)^2 + 2n + 4n + \log n + 50$
3. $f(n) = n + n^2 + 2n + n^4$
4. $f(n) = e^n + 5^{\log n}$

Problem 6.5 (Power)**.**

1. Consider the following implementation of `power`.

```
let rec power (x : int) (n : int) : int =
    if n = 0 then 1
    else x * power x (n - 1) ;;
```

   Define the worst-case time complexity of this function in terms of n.

2.
```
let rec power (x : int) (n : int) : int =
    if n = 0 then 1
    else if n mod 2 = 0 then power x (n / 2) * power x (n / 2)
    else  power x (n / 2) * power x ((n / 2) + 1);;
```

   Define the worst-case time complexity of this implementation in terms of n.

[1] https://dl.ebooksworld.ir/motoman/JBL.Foundations.Of.Algorithms.5th.Edition.www.EBooksWorld.ir.pdf

3. Rewrite `power` from Part 2 to improve the worst-case time complexity. What is the asymptotic runtime of the function right now?

Problem 6.6 (Minimum)**.** In lab 10, we defined a function `find_min` that finds the minimum of the list.

Listing 11: find_min

```
let rec find_min (xs : int list) : int =
  let rec split (xs : 'a list) =
    | [] -> [], []
    | [x] -> [x], []
    | first :: second :: rest -> let rest1, rest2 = split rest in
                                 first :: rest1, second :: rest2 in

  match xs with
  | [] -> raise (Invalid_argument "Empty List")
  | [x] -> x
  | _ -> let xs1, xs2 = split xs in
         min (find_min xs1) (find_min xs2) ;;
```

Derive the asymptotic runtime of the `find_min` function.