Mutating Algorithm on Immutable Data

1st Nassouh Al-Olabi
dept. name of organization (of Aff.)
name of organization (of Aff.)
Damascus, Syria
email address or ORCID

2th Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

***Abstract*—Often times engineers roll out functional programming in favor of an Object Oriented - Imperative style because of immutability due to inefficiency especially when their model is relatively large and in place mutations seem like the only efficient way to model changes or when implementing standard algorithms with in place mutations.**

***Index Terms*—immutable data structures, functional programming, algorithm design**

I. Introduction

In the ever growing field of computer science, and complex system design we find ourselves constantly devising new strategies for writing down a simple elegant solution, model - representation for our complex problems. At first during the low level C, Fortran days we'd write down naked data structs and write down procedures for processing them. then came the age of OO were we'd abstract and encapsulate data and it's behavior into Objects which served as both an api to interface with the data and a type for it and a namespace for methods that implicitly take this object as a parameter. this methodology has dominated the industry for over 40 years which is too long for any solution in this industry. The increasing need for distributed, stateless, fault-tolerant, concurrent, datacentric systems makes OO/imperative solutions harder to obtain and reason about.

II. Methods

A. Shadowing

Shadowing is a technique that could represent a changing variable. for instance, an accumulator. the technique is simple and possible in almost every programming language out there. at it's simplest form shadowing looks like this:

```
// Scala
object Main {
   def main(args: Array[String]){
      val i = 1;
      {
         val i = 2;
         {
            val i = 3;
         }
      }
   }
}
```

But that's not really useful and even more confusing and very error prone and I'd agree shadowing in of it's self isn't really useful but it's really at the core of any recursive solution since every function come with it's own scope and

```
def factorial (n: BigDecimal):BigDecimal =
    {
   if (n <= 1) 1
   else n * factorial(n-1)
}
```

Notice here n value range over {n, ... , 1} but it's not really changing each n deffer from the other and has it's own scope if you run this code it'll only go so far (around n = 9613 for this example) until you get a StackOverflowError... not good.

B. Tail Recursion

Tail recursion is when you simply return the value of a function call at the end (tail) of your function, in other words your functions has done it's job and handing over the rest of the work to another function

```
def factorial (n: BigDecimal):BigDecimal = {
  def helper(n: BigDecimal, Acc:
      BigDecimal): BigDecimal = {
    if (n <= 1) Acc
    else helper(n - 1, n * Acc)
  }
  helper(n, 1)
}
```

Notice that as n takes the values {n, n-1, n-2, ...} the accumulator also changes {n, n*(n-1), n*(n-1)*(n-2), ...} which is very similar to a for loop accumulator pattern, so similar that sometimes compilers compile it to an actual loop