Immutability for Better State Management

1st Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
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*—Recent growing demand for fault-tolerant, scalable, distributed systems has made some mainstream software architectures and patterns obsolete or rather harder to come by, and thus came the rise of stateless and functional solutions based on data immutability which has already been the cornerstone of ”Big Data”**

***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 programs in a procedural way explicitly telling the computer what to do in each step. 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. and here is where FP (functional programming) fits in. contrary to what you might think it's not the new kid on the block FP is as old as the first computers with lambda calculus as it's foundation FP learns a lot from it's mathematical roots in many aspects mainly with it being declarative and strict which allows for building outstanding compilers to understand it's concise declarative expressions. but in the old day computers couldn't afford the luxury of immutable data structures and so in place mutations encapsulated within a class was the most reasonable way to go about doing things. these days memory is so cheep you can always afford to store the value and a reference to it and it's hash and a couple of almost identical copies

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
// 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

```scala
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. Reassignment

TODO: choose which one

TODO: Paper definition: there's a subtle difference, well hidden behind the overloaded use of the symbol '=', that really sets the two apart. In the imperative program, '=' refers to a destructive update, assigning a new value to the left-hand-side variable, whereas in the functional program '=' means true equality, and that both the left-hand-side and the right-hand-side can be used interchangeably. This characteristic of functional programming (known as referential transparency or purity) has a profound influence on the way programs are constructed and reasoned about.

TODO: My definition: before we move on it's important to do the distinction between mutation and Reassignment. simply put when you mutate data the old version of it would become unusable and simple would cease to exist. On the other hand updating a variable should keep the old version usable. a perfect example of this is a VCS (version control system) it gives you the feeling that you are mutating files while in fact it's storing changes (updates) and updating a "HEAD" value to represent the last change (update)

C. *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

D. *Pure Functions*

Pure Functions when implemented correctly serves as a (possibly infinite) lookup table mapping from one type to another since variable x will always be the same $f(x)$ will too. this is what's known as referential transparency.

```
  #include <functional>
  #include <iostream>
  int sum(const int v[], const int& n) {
     std::function<int(int, int)> helper =
         [&v, &n, &helper](int index, int
         Acc) {
        if (index >= n) return Acc;
        return helper(index + 1, Acc +
            v[index]);
     };
     return helper(0, 0);
  }
  int main(int, char**) {
     const int a[] = {1, 2, 3, 4, 5, 6};
     // a[1] = 3; not allowed
     int total = sum(a, 6); // 21
     someFunction(a);
     otherFunction(a);
     if (total == sum(a,6))
        std::cout<<"It should be equal same
            function same argument?!";
     return 0;
  }
```

generally speaking "$someFunction$" and "$otherFunction$" could've done al sorts of things with a (changing an element value, adding more elements, removing some elements, delete the pointer entirely ...) but if they were pure functions or like in this case using a some sort of a language guarantee (here it's $const$) a will not be modified and in turns $total == sum(a, 6)$ and overall our program would be easier to reason about.

E. *Laziness*

sometimes refered to as call-by-need, it's the notion that "if data won't change and functions won't neither so would results" meaning that we wouldn't perfome any operations unless they're absolutely necessary

```
  def from(n: Int): LazyList[Int] =
     n #:: from(n+1)
     // here from(n+1) won't be evaluated
  def sieve (s: LazyList[Int]):
      LazyList[Int] =
     s.head #:: sieve(s.tail.filter(_ %
         s.head != 0))
  // nor would sieve(s.tail.filter(_ %
      s.head != 0))
  val primes = sieve(from(2))

  primes
     .take(10)
     .toList // now it's necessary
```

this behavior is implemented in some languages is what's known as Function0 that is "a function without parameters" but this would be somewhat expensive since you have to pack (copy) the closure of the function with it to achieve predictable behavior contrary to just keeping references. but if the data is immutable it would make sense to only pack references this is what's refereed to as laziness

F. *logfilesystem*

sth

G. *Append Logs*

sth

H. *Copy-On-Write*

sth

I. *Structural Sharing*

sth

J. *MVC*

sth

K. *elm*

sth

L. *Cpp example*

$https : //www.youtube.com/watch?v = y_m0ce1rzRI\&t = 4081s$