Complex State Management with Immutability

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**

CONTENTS

I Introduction 1

II Methods 2
- II-A Shadowing . . . 2
- II-B Reassignment . . . 2
- II-C Tail Recursion . . . 3
- II-D Pure Functions . . . 3
- II-E Laziness . . . 3
- II-F logfilesystem . . . 3
- II-G Append Logs . . . 3
- II-H Copy-On-Write . . . 3
- II-I Structural Sharing . . . 3
- II-J MVC . . . 3
- II-K elm . . . 3
- II-L Cpp example . . . 3

I. INTRODUCTION

When something is immutable, we say that it cannot be changed or that it is unchangeable. The definition of an immutable object in an object-oriented programming language is an object that has a state that cannot be mutated once instantiated (created). Moreover, an immutable class is a class whose instances cannot be mutated, meaning that there are no methods in the class that can mutate an instance of it. Classes that are not immutable can, however, have instances of it that are immutable.

The concept of immutability is important and utilizing immutable data is considered to ease software development and reasoning about programs in numerous ways, for example:

1. Predictability It is harder to understand and reason about programs that have shared mutable states with unclear interactions. Tracking mutations and maintaining the correct state of a program can be difficult. Using immutable data naturally, avoids state changes and forces the programmer to let data flow and be utilized in a different way throughout the program, making the state of the program more predictable. For example, calling the same function twice would yield the same result, and the outcome is predictable. Without the immutability guarantee, the second call could yield another result because of an underlying state mutation making it less predictable.

2. Testability Because immutable data can only be changed once during construction, they are inherently simpler and easier to unit test. One may reason that by restricting the number of possible mutations in a program; the number of potential errors of the program is also reduced. Testing is essentially to validate that mutations in the program occur correctly and thus having more mutations would require more testing. By restricting the number of mutations in a program, the program has fewer reasons for errors to occur and there are fewer state transitions to test.

3. Concurrency Immutable data are thread-safe, as data cannot mutate, there is no danger in having multiple threads access the same data at the same time and have synchronization issues.

4. Modularity Without depending on a local or global state, immutable types and data may be reused in different contexts more easily.

There are different varieties of immutability used in practice today, which includes:

Object immutability

An immutable object is an object that cannot be modified (mutated), i.e., its state cannot be mutated.

Class immutability

When every instantiated object of a class is immutable, then we say that the class itself is immutable.

Deep and shallow immutability (transitivity)

Immutability can be deep or shallow, i.e., transitive or non-transitive:

• **Deep (transitive)**

immutability means that all objects referred to by an immutable object must also be immutable.

• **Shallow (non-transitive)**

immutability has a more relaxed constraint, and the immutable object may refer to objects that are mutable, but the fields of the immutable object itself cannot be mutated.

Reference immutability (read-only references)

Languages with the support of reference immutability have the notion of read-only references. A readonly reference cannot mutate the object it is referring to. When all references to an object are readonly, then nothing can mutate the object, and the object is immutable. There are, however, often no guarantee that a referred to object is immutable because the object may still be mutated by some other reference that is not read-only, unless there is some analysis that ensures that there only exist readyonly references to that object. Reference immutability thus often ensure shallow (non-transitive) immutability and "reference immutability" does not mean that a reference itself is immutable.

Non-assignability

Non-assignability is a form of immutability and property of a variable that makes sure that it cannot be reassigned. Since fields of objects are variables and if no variable can be reassigned after its initial assignment, the object is effectively immutable if what the fields are referring to is immutable. This make non-assignability give shallow (non-transitive) immutability too. Assignment of an object's field mutates the object, but it does not mutate what was previously on the field or what was assigned to the field.

Concrete and abstract immutability

Concrete immutability does not allow any change to the object's state. Abstract immutability, however, allows an immutable object to change its internal state but not the object's "abstract" value. The object is still immutable from the perspective of the object's observer that can only see the abstract value, but the object may mutate internally. This can, for example, be useful to speed up certain operations, lazy initialization and buffering.

The immutability properties are:

• Mutable: We give a template the mutable property if an instance of that template can be mutated directly or indirectly.

• Shallow immutable: A template has the shallow (non-transitive) immutable property if the template does not have fields that can be reassigned, but has references to other objects that may be mutated (are shallow immutable or mutable).

• Deeply immutable: A template is deeply (transitive) immutable if all instances of that template cannot be mutated after initialization.

• Conditionally deeply immutable: The conditionally deeply immutable property is given to a template that is deeply immutable but depends on some other potentially mutable type. An example of this is a generic collection that can store different types, and the collection itself is declared in a way so that it cannot be mutated, but the type that is used with the collection may be shallow immutable or mutable.

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 block 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. 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 referred 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 referred to as laziness

F. logfilesystem

sth

G. Append Logs

sth

H. Copy-On-Write

sth

I. Structural Sharing

effecient cow reduce the overhead of threading and parallel processing

J. MVC

sth

K. elm

sth

L. Cpp example

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

References

[1]