Exercise 1 : Introduction to Concurrency

Freshly graduated from EPFL, you all have been hired as contractors for a successful and rapidly growing bank. The bank has recently been experiencing problems with their money management system, coded in Scala, and so they hired the best and brightest young engineers they could find: you! The system has been working perfectly fine so far, they tell you. In the past days, due to an increased number of customers, they had to switch from a single threaded sequential execution environment to a multithreaded concurrent environment, in which multiple threads may perform transactions concurrently. That's when problems started, your manager says…

Below is the code responsible to withdraw money from the account `from` and transfer it to the account `to`, within the same bank.

```
def transfer(from: Account, to: Account, amount: BigInt) {
  val balanceFrom = from.balance
  if (balanceFrom >= amount) {
    from.balance = balanceFrom - amount
    val balanceTo = to.balance
    to.balance = balanceTo + amount
  }
}
```

For the bank, it is very important that the following two properties hold in any sequence of transfer transactions:

1. The balance of an account never goes below 0.
2. The total sum of money held by the bank is constant.

Question 1

Does the above `transfer` method respect the two properties in a **sequential** execution environment ?

Yes.

Question 2

What can go wrong in a setting where multiple threads can execute the `transfer` method concurrently ? For each of the two desired properties of the system, check if its holds in this concurrent environment. If not, come up with an example execution which exhibits a violation of the property.

*Property 1 holds**

Assuming that the execution of two concurrent threads only interleaves instructions and that reads and writes are executed atomically, it can be shown that property 1 always holds. Unfortunately, such strong guarantees are not offered by the Java Memory Model. If you are interested, have a look at the note below on the Java Memory Model.

Violation of property 2

Consider 2 threads that execute concurrently `transfer(from, to, amount)` with the exact same parameters. Assume that the account `from` has sufficient funds for at least one transfer.

Thread 1 executes until it has computed the value `balanceFrom - amount` and then stops. Thread 2 then executes in its entirety the call to `transfer(from, to, amount)`. Then thread 1 resumes its execution and completes the call to `transfer`.

At the end of this execution, the total amount of money held by the bank has changed. It is has in fact increased by the value `amount`.

Note on the Java Memory Model

Assuming the Java Memory Model, both of the two properties can potentially be violated. Indeed, the model only ensure that the execution of each thread appears sequential to the thread itself, and not to any other concurrently running threads. Seemingly atomic instructions can be arbitrarily decomposed by the underlying virtual machine. Sequences of instructions can also be reordered at will by the VM, as long as the execution of a single thread appears as if it were executed sequentially. In this settings, both properties can be violated.

Question 3

For each of the proposed implementations of `transfer` below, check which of the properties hold. Additionally, check if the system is vulnerable to *deadlocks*.

Variant 1

```
def transfer(from: Account, to: Account, amount: Long) {
  val balanceFrom = from.balance
  if (balanceFrom >= amount) {
    from.synchronized {
      from.balance = balanceFrom - amount
    }
    to.synchronized {
      val balanceTo = to.balance
      to.balance = balanceTo + amount
    }
  }
}
```

*In this variant, property 2 can be violated. It is **not** vulnerable to deadlocks.*

Variant 2

```
def transfer(from: Account, to: Account, amount: Long) {
  from.synchronized {
    val balanceFrom = from.balance
    if (balanceFrom >= amount) {
      from.balance = balanceFrom - amount
      to.synchronized {
        val balanceTo = to.balance
        to.balance = balanceTo + amount
      }
    }
  }
}
```

In this variant, none of the two properties can be violated. However, it is susceptible to deadlocks.

Variant 3

```
object lock // Global object.
def transfer(from: Account, to: Account, amount: Long) {
  lock.synchronized {
    val balanceFrom = from.balance
    if (balanceFrom >= amount) {
      from.balance = balanceFrom - amount
      val balanceTo = to.balance
      to.balance = balanceTo + amount
    }
  }
}
```

In this last variant, none of the two properties can be violated and no deadlock can occur. It is however still not entirely satisfactory, since no two threads can execute transfers in parallel, even when the accounts are totally disjoint. Can you think of a better solution?

Exercise 2 : Parallel Reductions

Question 1

As a group, write a function called `minMax`, which should take a non-empty array as input and return a pair containing the smallest and the largest element of the array.

```
def minMax(a: Array[Int]): (Int, Int) = ???
```

Now write a parallel version of the function. You may use the constructs `task` and/or `parallel`, as seen in the lectures.

A solution:

```
def minMax(a: Array[Int]): (Int, Int) = {
  val threshold = 10

  def minMaxPar(a: Array[Int], from: Int, until: Int): (Int, Int) = {

    if (until - from <= threshold) {
      var i = from
      var min = a(from)
      var max = a(from)

      while (i < until) {
        val x = a(i)
        if(x < min) min = x
        if(x > max) max = x
        i = i + 1
      }

      (min, max)
    }
    else {
      val mid = (from + until) / 2
      val ((xMin, xMax),
           (yMin, yMax)) = parallel(minMaxPar(a, from, mid),
                                    minMaxPar(a, mid, until))
      (min(xMin, yMin), max(xMax, yMax))
    }
  }

  minMaxPar(a, 0, a.size)
}
```

Question 2

Imagine that the data structure you are given, instead of an `Array[A]`, is one called `ParSeq[A]`. This class offers the two following methods, which work in parallel:

```
def map[B](f: A => B): ParSeq[B]
def reduce(f: (A, A) => A): A
```

Can you write the following `minMax` function in terms of `map` and/or `reduce` operations ?

```
def minMax(data: ParSeq[Int]): (Int, Int) = ???
```

A solution:

```
def minMax(data: ParSeq[Int]): (Int, Int) = data.map({
  (x: Int) => (x, x)
}).reduce({
  case ((mn1, mx1), (mn2, mx2)) => (min(mn1, mn2), max(mx1, mx2))
})
```

Question 3

What property does the function `f` passed to `reduce` need to satisfy in order to have the same result regardless on how `reduce` groups the applications of the operation f to the elements of the data structure? Prove that your function `f` indeed satisfies that property.

The function `f` must be associative. That is, for any x, y, z, it should be the case that:

$$f(x, f(y, z)) == f(f(x, y), z).$$

Both the `min` and `max` functions are associative. In addition, it can be easily shown that pairwise application of associative functions is also associative. From this follows that `f` is indeed associative.

~~Exercise 3 : Depth~~

This question has been moved to the second exercise session.