TDT4165 - Programming Languages

Scala Project

Benjamin Zubača

October 29, 2024

Part I: Introduction to Scala

Task 1: Scala Introduction

a)

```scala
def generateArray(): Array[Int] =
    val arr = Array[Int](50)
    for i <- 1 to 50 do arr(i - 1) = i
    arr
```

b)

```scala
def sum(arr: Array[Int]): Int =
    var sum = 0
    for num <- arr do sum += num
    sum
```

c)

```scala
def recursiveSum(arr: Array[Int]): Int = arr match
    case Array() => 0
    case Array(head, tail*) => head + recursiveSum(tail.toArray)
```

d)

```scala
def fibonacci(n: BigInt): BigInt = n match
    case n if n == BigInt(0) => 0
    case n if n == BigInt(1) => 1
    case _ => fibonacci(n - 1) + fibonacci(n - 2)
```

In Scala, the `Int` data type serves as a wrapper around Java's 32-bit signed `int` primitive[1]. It can represent integers ranging from -2^{31} to $2^{31} - 1$. For mathematical operations involving larger values, Scala provides the `BigInt` type, which allows for arbitrary-precision integer arithmetic[2]. In other words, this data type effectively removes the upper and lower bounds on integer values that can be represented and manipulated.

[1] https://www.scala-lang.org/api/current/scala/Int.html#
[2] https://www.scala-lang.org/api/current/scala/math/BigInt.html#

Task 2: Higher-Order Programming in Scala

a)

```scala
def quadraticEquation(a: Float, b: Float, c: Float): (Boolean, Option[Double], Option[Double]) =
    val discriminant = b * b - 4.0 * a * c

    if (discriminant >= 0) {
      val x1 = (-b + math.sqrt(discriminant)) / (2.0 * a)
      val x2 = (-b - math.sqrt(discriminant)) / (2.0 * a)
      (true, Some(x1), Some(x2))
    } else (false, None, None)
```

```scala
def quadratic(a: Float, b: Float, c: Float): Float => Float =
    (x: Float) => a * x * x + b * x + c
```

b) For reference, the following code listings show the Oz implementations for tasks 1 and 4 from assignment 3:

```
proc {QuadraticEquation A B C}
   RealSol
   X1
   X2
   Discriminant = B*B - 4.0*A*C
in
   if Discriminant >= 0.0 then
      RealSol = true
      X1 = (~B + {Float.sqrt Discriminant}) / (2.0*A)
      X2 = (~B - {Float.sqrt Discriminant}) / (2.0*A)
   else
      RealSol = false
      X1 = X2 = noSol
   end
end
```

```
fun {Quadratic A B C}
   fun {$ X}
      A*X*X + B*X + C
   end
end
```

While the Scala and Oz programs appear quite similar, there are some differences worth mentioning. For function declarations, Scala uses the keyword `def` to define both functions and procedures. In contrast, Oz differentiates between them, using `proc` for procedures and `fun` for functions.

Another distinction is in the handling of the return values. In the Scala implementation, we return a tuple with a `Boolean` and two `Option[Double]` values from the `quadraticEquation` function. On the other hand, the Oz `QuadraticEquation` procedure uses multiple output parameters. Optional values are represented differently as well: In Scala, we use `Option[Double]` to represent potentially undefined results, while in Oz, we use the atom `noSol` for the same purpose.

For mathematical operations, in Scala, we rely on `math.sqrt` for square root calculations, while in Oz, we make use of `Float.sqrt`. Both languages support higher-order functions, but their syntax varies: In Scala, we return the anonymous function within the `quadratic` function using `=>`, whereas in Oz, we achieve the same in the `Quadratic` function using the `fun {$ X}` construct.

As for the type system, in Scala, we explicitly declare types like `Float` and `Double` in the function definitions, reflecting the static typing system. In Oz, we rely on dynamic typing, with no explicit type declarations.

Task 3: Concurrency in Scala

a)

```
def createThread(fn: () => Unit): Thread = Thread(() => fn())
```

Here, we assume that the input function that the thread receives takes zero arguments.

b)
- This code simulates a concurrent system where two threads continuously transfer "units" from `value1` to `value2` within the singleton object `ConcurrencyTroubles`. When `value1` reaches `0`, both fields are reset: `value1` to `1000` and `value2` to `0`. The system aims to maintain a constant sum of `1000` between `value1` and `value2`. This sum is continuously updated and printed, along with the individual values of `value1` and `value2`.
- The code is not working as expected. Specifically, the lack of synchronization mechanisms for the operations on `value1` and `value2` leads to inconsistent updates. As a result, the sum can be incorrect, occasionally exceeding the expected total of `1000` before the values are reset.
- Several issues are likely occurring in this code. Firstly, there are race conditions, as multiple threads are accessing and modifying the shared variables (`variable1`, `variable2` and `sum`) without proper synchronization. Secondly, the methods `moveOneUnit()` and `updateSum()` are not atomic, which can lead to inconsistencies when executed concurrently. Lastly, the threads can interleave their executions in unpredictable ways, causing the sum to be incorrect.
- Oz, a multi-paradigm language, could potentially exhibit or avoid the concurrency issues seen in the Scala code. Its support for data flow variables and single-assignment can prevent many of these problems. However, if a programmer deliberately attempted to recreate this scenario in Oz using shared state and without proper synchronization, similar issues could arise.
- This behavior could significantly impact a real-world banking system. Consider a scenario where this code represents a simplified bank account transfer system where `value1` is the balance of an account A, `value2` is the balance of an account B, and `moveUnit()` represents a transfer of $1 from A to B. The race conditions and lack of synchronization could lead to lost or duplicated transfers, incorrect account balances, and inconsistent total bank assets represented by `sum`.

c) To ensure thread safety, we can modify the code using two approaches. The first method employs Scala's `synchronized` blocks:

```
object ConcurrencyTroubles:
  private var value1 = 1000
  private var value2 = 0
  private var sum = 0

  private def moveOneUnit(): Unit = synchronized {
    value1 -= 1
    value2 += 1
    if (value1 == 0)
      value1 = 1000
```

```scala
      value2 = 0
  }

  private def updateSum(): Unit = synchronized {
    sum = value1 + value2
  }

  private def execute(): Unit =
    while true do
      moveOneUnit()
      updateSum()
      Thread.sleep(100)

  @main def runThreads(): Unit =
    for (i <- 1 to 2) do
      val thread = Thread(() => execute())
      thread.start()

    while true do
      updateSum()
      println(s"$sum [$value1] [$value2]")
```

The `moveOneUnit()` and the `updateSum` methods are now synchronized, ensuring that only one thread can execute them at a time. This prevents race conditions and guarantees that operations on `variable1`, `variable2`, and `sum` are atomic from the perspective of other threads.

Another way to ensure thread safety is by using Java's `AtomicReference`:

```scala
import java.util.concurrent.atomic.AtomicReference

object ConcurrencyTroubles:
  private case class State(value1: Int, value2: Int)

  private val state = AtomicReference(State(1000, 0))
  private val sum = AtomicReference(0)

  private def moveOneUnit(): Unit =
    state.getAndUpdate { s =>
      if (s.value1 == 0) State(1000, 0)
      else State(s.value1 - 1, s.value2 + 1)
    }

  private def updateSum(): Unit =
    val currentState = state.get()
    sum.set(currentState.value1 + currentState.value2)

  private def execute(): Unit =
    while true do
      moveOneUnit()
```

```
      updateSum()
      Thread.sleep(100)

  @main def runThreads(): Unit =
    for (i <- 1 to 2) do
      val thread = Thread(() => execute())
      thread.start()

    while true do
      updateSum()
      val currentState = state.get()
      val currentSum = sum.get()
      println(s"$currentSum [${currentState.value1}] [${currentState.value2}]")
```

Part II: The Banking System

Task 1: Preliminaries

1.1 Implementing the TransactionPool

```scala
import scala.collection.mutable

class TransactionPool:
  private val transactions = mutable.Queue[Transaction]()

  def remove(t: Transaction): Boolean = this.synchronized {
    val initialSize = size
    transactions.removeAll(_ == t)
    initialSize != size
  }

  def isEmpty: Boolean = this.synchronized {
    transactions.isEmpty
  }

  def size: Integer = this.synchronized {
    transactions.size
  }

  def add(t: Transaction): Boolean = this.synchronized {
    transactions.enqueue(t)
    true
  }

  def iterator: Iterator[Transaction] = this.synchronized {
    transactions.iterator
  }
```

This implementation of `TransactionPool` uses a mutable `Queue` as the underlying data structure to hold the transactions. Thread safety is achieved using Scala's `synchronized` keyword on the public methods: The `this.synchronized` restricts1 method execution to a single thread at a time, thus preventing concurrent modifications that might corrupt the data structure.

1.2 Account Functions & 1.3 Eliminating Exceptions

```
class Account(val code: String, val balance: Double):
  def withdraw(amount: Double): Either[String, Account] =
    if (amount < 0) Left("Invalid amount: withdrawal amount must be positive")
    else if (amount > balance) Left(s"Insufficient funds: cannot withdraw $amount from balance $balance")
    else Right(Account(code, balance - amount))

  def deposit(amount: Double): Either[String, Account] =
    if (amount < 0) Left("Invalid amount: deposit amount must be positive")
    else Right(Account(code, balance + amount))
```

This implementation of the `Account` class models an immutable account by defining `withdraw` and `deposit` methods that return a new `Account` instance with the updated balance.

To handle invalid transactions without exceptions, both methods return an `Either` type. They validate the transaction and, in case of an error, return a `Left` with an appropriate message. If the transaction is valid, they return a `Right` containing a new `Account` instance with the updated balance.

Task 3: Explain How the Code Works

1. The bank system code consists of three main components: `Account`, `Transaction`, and `Bank`, each serving distinct purposes to maintain data integrity, support concurrency, and handle errors gracefully.

   The `Account` class represents an individual bank account identified by a unique code and a balance. Each transaction involving an account, whether a withdrawal or deposit, returns a new `Account` instance with an updated balance instead of modifying the original instance. The `withdraw` and `deposit` methods leverage the `Either` type to eliminate exceptions; they return a `Left` containing an error message for invalid amounts and a `Right` containing a new `Account` instance when successful.

   The `Transaction` class models a transaction between two accounts. Each transaction includes details about the sender and receiver account codes, the transfer amount, and the maximum allowed retry attempts. The class maintains a `status` field with three possible states - `PENDING`, `SUCCESS`, or `FAILED` - and provides synchronized methods to safely update this status across multiple threads. When a transaction fails, it is retried up to the specified limit. If retries are still available, the status resets to `PENDING` automatically, allowing the transaction to reattempt completion.

   The `Bank` class acts as the system's central manager, handling account creation, balance retrieval, and transaction processing. It uses a thread safe account registry and two `TransactionPool` queues to organize pending and completed transactions. Through the `transfer` method, new transactions are added to the pool, and the `processTransactions` method manages the concurrent processing of each transaction in a separate thread. Once a transaction completes successfully, it moves to the completed transactions pool. If a transaction fails, it will retry up to the retry limit; if it continues to fail, it will also be moved to the completed pool.

2. The `Account` class was the simplest to implement because its immutable design eliminates the need for explicit synchronization, which would otherwise be necessary if the class was mutable. Having experience with other functional languages that handle concurrency through immutability made this approach feel straightforward to me.

3. The `Bank` class was probably the most challenging. One difficulty was designing the `accountsRegistry` map as an immutable `val` field while still supporting atomic updates. I also encountered an `IndexOutOfBoundsException` in tests 11 and 12, with the error message `100 is out of bounds (min 0, max 99)`. The stack trace didn't reveal the root cause clearly, but I eventually traced it to a likely issue with modifying the `TransactionPool` queue during iteration in `processTransactions`. To fix this, I cloned the queue with `toList` and iterated over the clone, which likely prevented issues from concurrent modifications.

4. Refer to the answer for question 1.