Scala Project Part 1 TDT4165

Mehdi Mahmoud, Kristoffer Fredriksen

October 2025

Task 1: Scala Introduction

Task 1d

The `BigInt` data type is intended for use when integer values might exceed the range that is supported by the int data type.

Task 2: Higher-Order Programming in Scala

Assignment 3 Task 1: Quadratic Equation Solver

Oz implementation:

```
proc {QuadraticEquation A B C ?RealSol ?X1 ?X2}
   D = B*B - 4*A*C
   if D < 0 then
      RealSol = false
   else
      RealSol = true
      X1 = (~B + {Sqrt D}) / (2*A)
      X2 = (~B - {Sqrt D}) / (2*A)
   end
end
```

Scala implementation:

```
def quadraticEquation(a: Double, b: Double, c: Double): (Boolean, Double, Double) = {
  val d = b * b - 4 * a * c
  if (d < 0) (false, Double.NaN, Double.NaN)
  else {
    val x1 = (-b + Math.sqrt(d)) / (2 * a)
    val x2 = (-b - Math.sqrt(d)) / (2 * a)
    (true, x1, x2)
  }
}
```

Key differences:

- **Variable binding:** Oz uses logic variables (`?X1`), bound later. Scala uses return tuples (`(Boolean, Double, Double)`).
- **Procedural model:** Oz separates procedures (no return) and functions (with return). Scala only uses functions.
- **Mutability:** Oz variables are single-assignment; Scala distinguishes `val` (immutable) and `var` (mutable).
- **Typing:** Oz is dynamic, Scala is static.
- **Output:** Oz passes output via reference arguments; Scala returns composite values.

Assignemnt 3 Task 4: Returning a Function

Oz implementation:

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

{System.show {{Quadratic 3 2 1} 2}}  % prints 17
```

Scala implementation:

```
def quadratic(a: Double, b: Double, c: Double): Double => Double = {
  (x: Double) => a * x * x + b * x + c
}

println(quadratic(3, 2, 1)(2))  // prints 17
```

Key differences:

- **Syntax:** Oz uses {...} for calls and `fun/end`; Scala uses parentheses and braces.
- **Typing:** Oz is dynamically typed, while Scala is statically typed and must specify return type `Double => Double`.
- **Functions as values:** Both support higher-order functions, but Scala represents them explicitly as objects of type `Function1[Double, Double]`.
- **Return rules:** Scala returns the last expression automatically, while Oz binds results through logic variables.

Summary: In Scala, all computations are expressions that return values, and types must be declared. Functions are first-class typed objects. In Oz, computations rely on binding of logic variables and procedural abstractions; procedures and functions are distinct.

Task 3 Concurrency Troubles

1. What is this code supposed to do?

The program is designed to maintain the invariant:

$$\text{sum} = value1 + value2 = 1000$$

Two threads repeatedly execute the function `moveOneUnit()`, which transfers one unit from `value1` to `value2`. Meanwhile, the main thread continuously prints the current values of `sum`, `value1`, and `value2`. When `value1` reaches zero, both variables are reset (`value1 = 1000`, `value2 = 0`).

In theory, every printed line should display a constant sum of 1000:

```
1000 [999 1]
1000 [998 2]
1000 [997 3]
...
```

2. Is it working as expected?

No. The output often shows inconsistent values and irregular jumps in the sequence. While the printed sum may still show 1000 by coincidence, the state of `value1` and `value2` does not behave deterministically.

The code is not thread-safe and therefore does not behave as intended.

3. What is happening?

The program suffers from a **race condition**. Both `moveOneUnit()` and `updateSum()` access and modify shared variables (`value1`, `value2`, `sum`) **without synchronization**.

This leads to interleaving operations such as:

- Thread A reads `value1 = 500`.
- Thread B decrements `value1` to 499 and increments `value2`.
- Thread A writes back outdated values, effectively overwriting Thread B's update.

Because these updates are not atomic, the program's internal state becomes inconsistent.

4. Would it be possible to experience the same behavior in Oz? Why or why not?

No, not in the same way. Oz uses **dataflow variables** and **single-assignment semantics**, which means a variable can only be bound once. Concurrency in Oz is based on message passing and logical dataflow rather than shared mutable state.

This design eliminates the kind of race conditions seen in the Scala example, since no two concurrent threads can modify the same variable.

5. Example of impact in a real application

A similar race condition in a real system could cause serious logical or financial errors. Examples include:

- Banking software losing or duplicating money when multiple transfers occur simultaneously.
- Inventory systems reporting incorrect stock quantities in e-commerce applications.
- Multithreaded servers overwriting cached values due to concurrent writes.