Beyond the Future with purely functional Scala

Ilya Murzinov

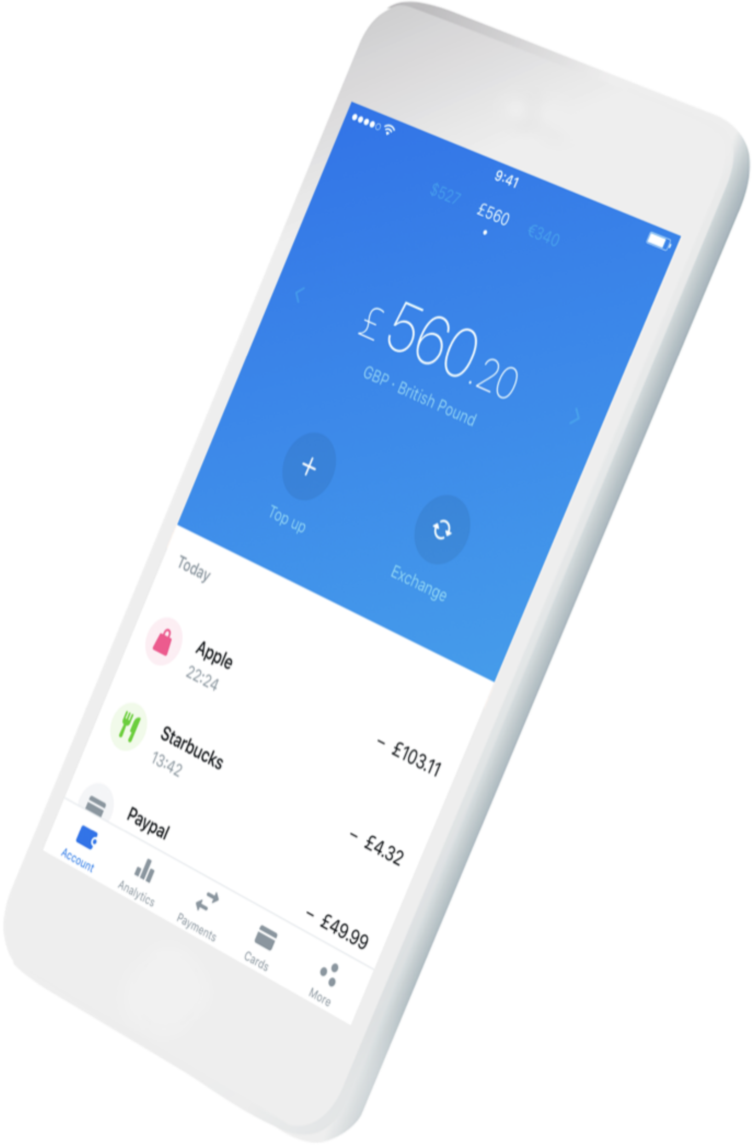
ilya-murzinov

ilyamurzinov

✉ murz42@gmail.com

Revolut

Beyond Banking



Typical ETL

```
def extract(): Seq[String] = ???
def transform(strings: Seq[String]): Seq[WTF] = ???
def load(wtfs: Seq[WTF]): Unit = ???
```

Typical ETL

```scala
def extract(): Seq[String] = ???
def transform(strings: Seq[String]): Seq[WTF] = ???
def load(wtfs: Seq[WTF]): Unit = ???
```

```scala
def etl(): Unit = {
  val strings = extract()
  val wtfs = transform(strings)
  load(wtfs)
}
```

Referential transparency

```
def goodFunction() = 2 + 2
```

Referential transparency

```
def goodFunction() = 2 + 2
```

```
val v1 = goodFunction() + goodFunction()

val goodResult = goodFunction()
val v2 = goodResult + goodResult
```

Referential transparency

```
def goodFunction() = 2 + 2
```

```
val v1 = goodFunction() + goodFunction()

val goodResult = goodFunction()
val v2 = goodResult + goodResult
```

```
v1 == v2 // true
```

Referential transparency

```
def badFunction() = {
  sendMessage()
  2 + 2
}
```

Referential transparency

```
def badFunction() = {
  sendMessage()
  2 + 2
}
```

```
val v3 = badFunction() + badFunction()

val badResult = badFunction()
val v4 = badResult + badResult
```

Referential transparency

```
def badFunction() = {
  sendMessage()
  2 + 2
}
```

```
val v3 = badFunction() + badFunction()

val badResult = badFunction()
val v4 = badResult + badResult
```

```
v3 == v4 // true-ish
```

Can we do better?

Yes we can!

Future FTW

```scala
import scala.concurrent.Future

val extractF: Future[Seq[String]] = ???
val transformF: Seq[String] => Future[Seq[WTF]] = ???
val loadF: Seq[WTF] => Future[Unit] = ???
```

Future FTW

```scala
import scala.concurrent.Future

val extractF: Future[Seq[String]] = ???
val transformF: Seq[String] => Future[Seq[WTF]] = ???
val loadF: Seq[WTF] => Future[Unit] = ???
```

```scala
val etlF: Future[Unit] = for {
  strings <- extractF
  wtfs <- transformF(strings)
  _ <- loadF(wtfs)
} yield ()
```

Future FTW

```
import scala.concurrent.Future

val extractF: Future[Seq[String]] = ???
val transformF: Seq[String] => Future[Seq[WTF]] = ???
val loadF: Seq[WTF] => Future[Unit] = ???
```

```
val etlF: Future[Unit] = for {
  strings <- extractF
  wtfs <- transformF(strings)
  _ <- loadF(wtfs)
} yield ()
```

```
> sbt compile
```

Oops!

```
[error] Main.scala:30:5: Cannot find an implicit ExecutionContext.
[error] You might pass
[error] an (implicit ec: ExecutionContext) parameter to your method
[error] or import scala.concurrent.ExecutionContext.Implicits.global.
[error]   _ <- loadF(wtfs)
[error]     ^
[error] Main.scala:29:8: Cannot find an implicit ExecutionContext
[error] You might pass
[error] an (implicit ec: ExecutionContext) parameter to your method
[error] or import scala.concurrent.ExecutionContext.Implicits.global.
[error]   wtfs <- transformF(strings)
[error]        ^
[error] Main.scala:28:11: Cannot find an implicit ExecutionContext
[error] You might pass
[error] an (implicit ec: ExecutionContext) parameter to your method
[error] or import scala.concurrent.ExecutionContext.Implicits.global.
[error]   strings <- extractF
[error]           ^
[error] three errors found
[error] (Compile / compileIncremental) Compilation failed
```

ExecutionContext

```scala
val etlF: Future[Unit] = for {
  strings <- extractF
  wtfs <- transformF(strings)
  _ <- loadF(wtfs)
} yield ()
```

ExecutionContext

```scala
val etlF: Future[Unit] = for {
  strings <- extractF
  wtfs <- transformF(strings)
  _ <- loadF(wtfs)
} yield ()
```

```scala
val etlF = extractF
  .flatMap(strings => transformF(strings))
  .flatMap(wtfs => loadF(wtfs))
```

ExecutionContext

```
val etlF: Future[Unit] = for {
  strings <- extractF
  wtfs <- transformF(strings)
  _ <- loadF(wtfs)
} yield ()
```

```
val etlF = extractF
  .flatMap(strings => transformF(strings))
  .flatMap(wtfs => loadF(wtfs))
```

```
def flatMap[S](f: T => Future[S])
              (implicit ec: ExecutionContext): Future[S] = ???
```

ExecutionContext

```
val etlF: Future[Unit] = for {
  strings <- extractF         // <- IO-bound
  wtfs <- transformF(strings) // <- CPU-bound
  _ <- loadF(wtfs)            // <- IO-bound
} yield ()
```

ExecutionContext

```scala
val etlF: Future[Unit] = for {
  strings <- extractF         // <- IO-bound
  wtfs <- transformF(strings) // <- CPU-bound
  _ <- loadF(wtfs)            // <- IO-bound
} yield ()
```

```scala
val comp = ExecutionContext.fromExecutor(
  Executors.newFixedThreadPool(
    Runtime.getRuntime.availableProcessors()))

val io = ExecutionContext.fromExecutor(
  Executors.newCachedThreadPool())
```

```scala
extractF
  .flatMap(strings => transformF(strings))(comp)
  .flatMap(wtfs => loadF(wtfs))(io)
```

Drawbacks of Future

Drawbacks of Future

- Eager (thus not ref. transparent)

Drawbacks of Future

- Eager (thus not ref. transparent)
- Not cancellable

Drawbacks of Future

- Eager (thus not ref. transparent)
- Not cancellable
- Always asyncronous

Drawbacks of Future

- Eager (thus not ref. transparent)
- Not cancellable
- Always asyncronous
- Memoized

Drawbacks of Future

- Eager (thus not ref. transparent)
- Not cancellable
- Always asyncronous
- Memoized
- Leaky API

Can we do even better?

Yes we can!

Monix

Monix modules

- `monix-eval` - Task, Coeval, MVar etc.
- `monix-reactive` - Observable, Observer (push-based streaming)
- `monix-tail` - Iterant (pull-based streaming)
- `monix-execution` - Scheduler & bunch of performance hacks

`Task[A]`

Benefits of Task

- Lazy (ref. transparent)

Benefits of Task

- Lazy (ref. transparent)
- Cancellable

Benefits of Task

- Lazy (ref. transparent)
- Cancellable
- Not always asyncronous

Benefits of Task

- Lazy (ref. transparent)
- Cancellable
- Not always asyncronous
- Never blocks threads

Benefits of Task

- Lazy (ref. transparent)
- Cancellable
- Not always asyncronous
- Never blocks threads
- Doesn't expose blocking API

Benefits of Task

- Lazy (ref. transparent)
- Cancellable
- Not always asyncronous
- Never blocks threads
- Doesn't expose blocking API
- Stack (and heap) safe

Benefits of Task

- Lazy (ref. transparent)
- Cancellable
- Not always asyncronous
- Never blocks threads
- Doesn't expose blocking API
- Stack (and heap) safe
- Not memoized by default

Scheduler

Can:

- Schedule delayed execution
- Schedule periodic execution
- Provide cancellation token
- Use different execution models

ExecutionModel

- `AlwaysAsyncExecution`
- `SynchronousExecution`
- `BatchedExecution`

Scheduler

```
Scheduler.computation(name = "my-computation")

Scheduler.io(name = "my-io")
```

Scheduler

```
Scheduler.computation(name = "my-computation")

Scheduler.io(name = "my-io")
```

```
Scheduler.fixedPool("my-fixed-pool", 10)

Scheduler.singleThread("my-single-thread")
```

Creating a task

```scala
import monix.eval.Task

// eagerly evaluates the argument
Task.now(42)
Task.now(println(42))

// suspends argument evaluation
Task.eval(println(42))

// suspends evaluation + makes it asynchronous
Task(println(42))

...

Task.evalOnce(...)
Task.defer(...)
Task.deferFuture(...)
Task.deferFutureAction(...)

...
```

Thread shifting

```
val t = Task.eval(println(42))

t.executeAsync

t.executeOn(io)

t.asyncBoundary(io)
```

Thread shifting

```scala
import monix.execution.Scheduler
import monix.execution.Scheduler.Implicits.global

lazy val io = Scheduler.io(name = "my-io")

val source = Task.eval(println(
  s"Running on thread: ${Thread.currentThread.getName}"))

val async = source.executeAsync
val forked = source.executeOn(io)

val onFinish = Task.eval(println(
  s"Ends on thread: ${Thread.currentThread.getName}"))

source // executes on main
  .flatMap(_ => source) // executes on main
  .flatMap(_ => async) // executes on global
  .flatMap(_ => forked) // executes on io
  .asyncBoundary // switch back to global
  .doOnFinish(_ => onFinish) // executes on global
  .runAsync
```

Composing tasks

```
val extractT: Task[Seq[String]] = ???
val transformT: Seq[String] => Task[Seq[WTF]] = ???
val loadT: Seq[WTF] => Task[Unit] = ???

val etl: Task[Unit] = for {
  strings <- extractT
  wtfs <- transformT(strings)
  _ <- loadT(wtfs)
} yield ()
```

Composing tasks

```
val extractT: Task[Seq[String]] = ???
val transformT: Seq[String] => Task[Seq[WTF]] = ???
val loadT: Seq[WTF] => Task[Unit] = ???

val etl: Task[Unit] = for {
  strings <- extractT
  wtfs <- transformT(strings)
  _ <- loadT(wtfs)
} yield ()
```

```
val extract1: Task[Seq[String]] = ???
val extract2: Task[Seq[String]] = ???
val extract3: Task[Seq[String]] = ???

val extract =
  Task.parMap3(extract1, extract2, extract3)(_ :+ _ :+ _)
```

Composing tasks

```
val comp = Scheduler.computation(name = "my-computation")
val io = Scheduler.io(name = "my-io")

val etl: Task[Unit] = for {
  strings <- extractT.executeOn(io)
  wtfs <- transformT(strings).executeOn(comp)
  _ <- loadT(wtfs).executeOn(io)
} yield ()
```

Composing tasks

```scala
val tasks: Seq[Task[A]] = Seq(task1, task2, ...)

// Seq[Task[A]] => Task[Seq[A]]
Task.sequence(tasks)

Task.gather(tasks)

Task.gatherUnordered(tasks)
```

Composing tasks

```
val tasks: Seq[Task[A]] = Seq(task1, task2, ...)

// Seq[Task[A]] => Task[Seq[A]]
Task.sequence(tasks)

Task.gather(tasks)

Task.gatherUnordered(tasks)
```

```
// Seq[Task[A]] => Task[A]
Task.raceMany(tasks)
```

Task cancellation

```
val task = ???

val f: CancelableFuture[Unit] = t.runAsync

f.cancel()
```

Task cancellation

```
import monix.execution.Scheduler.Implicits.global

val sleep = Task(Thread.sleep(100))

val t = sleep.flatMap(_ => Task.eval(println(42)))

t.doOnCancel(Task.eval(println("On cancel")))
  .runAsync
  .cancel()

Thread.sleep(1000)
```

Task cancellation

```scala
import monix.execution.Scheduler.Implicits.global

val sleep = Task(Thread.sleep(100))

val t = sleep.flatMap(_ => Task.eval(println(42)))

t.doOnCancel(Task.eval(println("On cancel")))
  .runAsync
  .cancel()

Thread.sleep(1000)
```

```
> sbt runMain demo.Main
On cancel
42
```

Task cancellation

```
import monix.execution.Scheduler.Implicits.global

val sleep = Task(Thread.sleep(100)).cancelable

val t = sleep.flatMap(_ => Task.eval(println(42)))

t.doOnCancel(Task.eval(println("On cancel")))
  .runAsync
  .cancel()

Thread.sleep(1000)
```

Task cancellation

```scala
import monix.execution.Scheduler.Implicits.global

val sleep = Task(Thread.sleep(100)).cancelable

val t = sleep.flatMap(_ => Task.eval(println(42)))

t.doOnCancel(Task.eval(println("On cancel")))
  .runAsync
  .cancel()

Thread.sleep(1000)
```

```
> sbt runMain demo.Main
On cancel
```

Task memoization

```
val t: Task = ???
t.memoize
t.memoizeOnSuccess
```

Task memoization

```scala
val t: Task = ???
t.memoize
t.memoizeOnSuccess
```

```scala
var effect = 0

val source = Task.eval {
  effect += 1
  if (effect < 3) throw new RuntimeException("dummy") else effect
}

val cached = source.memoizeOnSuccess
```

References

- Monix (https://monix.io)
- Monix vs Cats-Effect
- Scalaz 8 IO vs Akka (typed) actors vs Monix @ SoftwareMill
- Solution of the example (https://github.com/ilya-murzinov/seuraajaa)

Questions?

Thanks!