Monix in practice

Ilya Murzinov

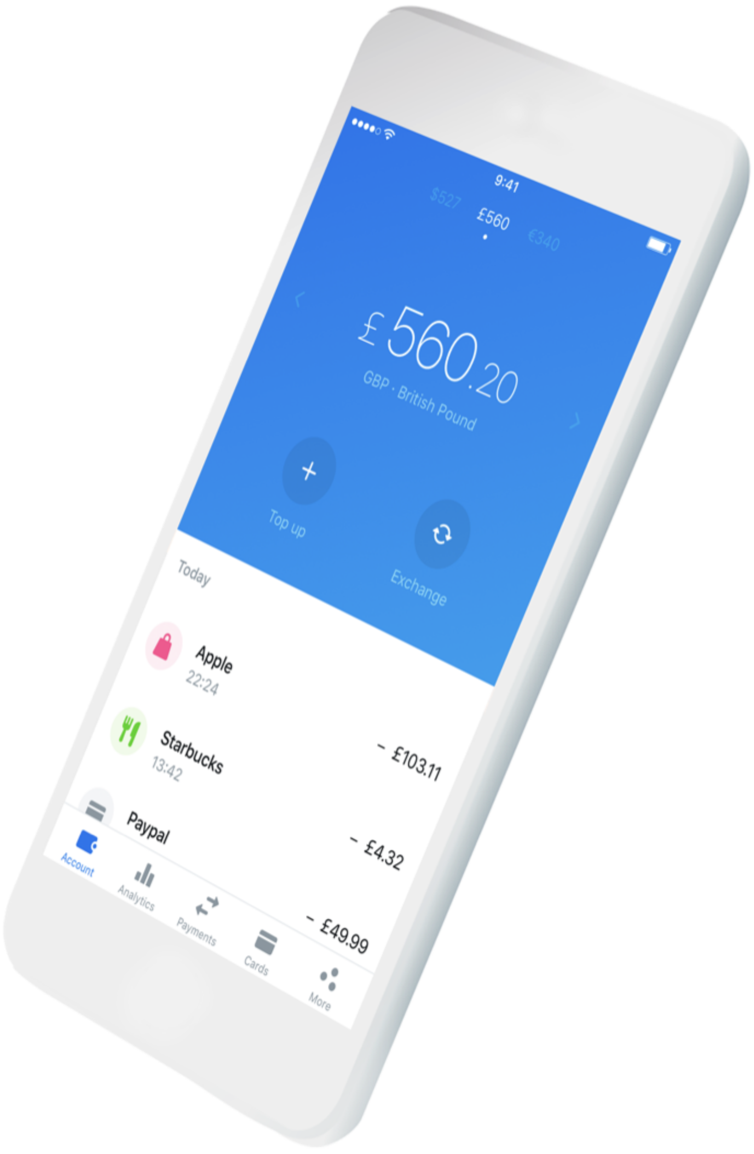
https://twitter.com/ilyamurzinov

https://github.com/ilya-murzinov

Slides: https://ilya-murzinov.github.io/slides/scalaspb2018.pdf

Revolut

Beyond Banking



Referential transparency

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

Referential transparency

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

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

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]
```

Task vs Future

Task vs Future

`scala.concurrect.Future`:

- Eager (thus not ref. transparent)

Task vs Future

`scala.concurrect.Future`:

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

Task vs Future

`scala.concurrect.Future`:

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

Task vs Future

`scala.concurrect.Future`:

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

Task vs Future

`scala.concurrect.Future`:

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

`monix.Task`:

- Lazy (ref. transparent)

Task vs Future

`scala.concurrect.Future`:

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

`monix.Task`:

- Lazy (ref. transparent)
- Cancellable

Task vs Future

`scala.concurrect.Future`:

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

`monix.Task`:

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

Task vs Future

`scala.concurrect.Future`:

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

`monix.Task`:

- Lazy (ref. transparent)
- Cancellable
- Not always asyncronous
- Stack (and heap) safe

Scheduler

- 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

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

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

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

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

Creating a task

```
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

```scala
val extract: Task[Seq[String]] = ???
val transform: Seq[String] => Task[Seq[WTF]] = ???
val load: Seq[WTF] => Task[Unit] = ???

for {
  strings <- extract
  transformed <- transform(strings)
  _ <- load(transformed)
} yield ()
```

Composing tasks

```
val extract: Task[Seq[String]] = ???
val transform: Seq[String] => Task[Seq[WTF]] = ???
val load: Seq[WTF] => Task[Unit] = ???

for {
  strings <- extract
  transformed <- transform(strings)
  _ <- load(transformed)
} 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 tasks: Seq[Task[A]] = Seq(task1, task2, ...)

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

Task.gather(tasks)

Task.gatherUnordered(tasks)
```

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)
```

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

Task cancellation

```scala
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.runAsync.cancel()

Thread.sleep(1000)
```

Task cancellation

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

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

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

t.runAsync.cancel()

Thread.sleep(1000)
```

Observable[A]

Observable[A]

- Lazy (ref. transparent)
- Cancellable
- Safe (doesn't expose unsafe or blocking operations)
- Allows fine-grained control over execution
- Models single producer - multiple consumers communication

Monix vs Akka streams

`Monix` has

- Simpler API
- Lighter (no dependency on actor framework)
- Better execution control
- Easier to understand internals
- Faster

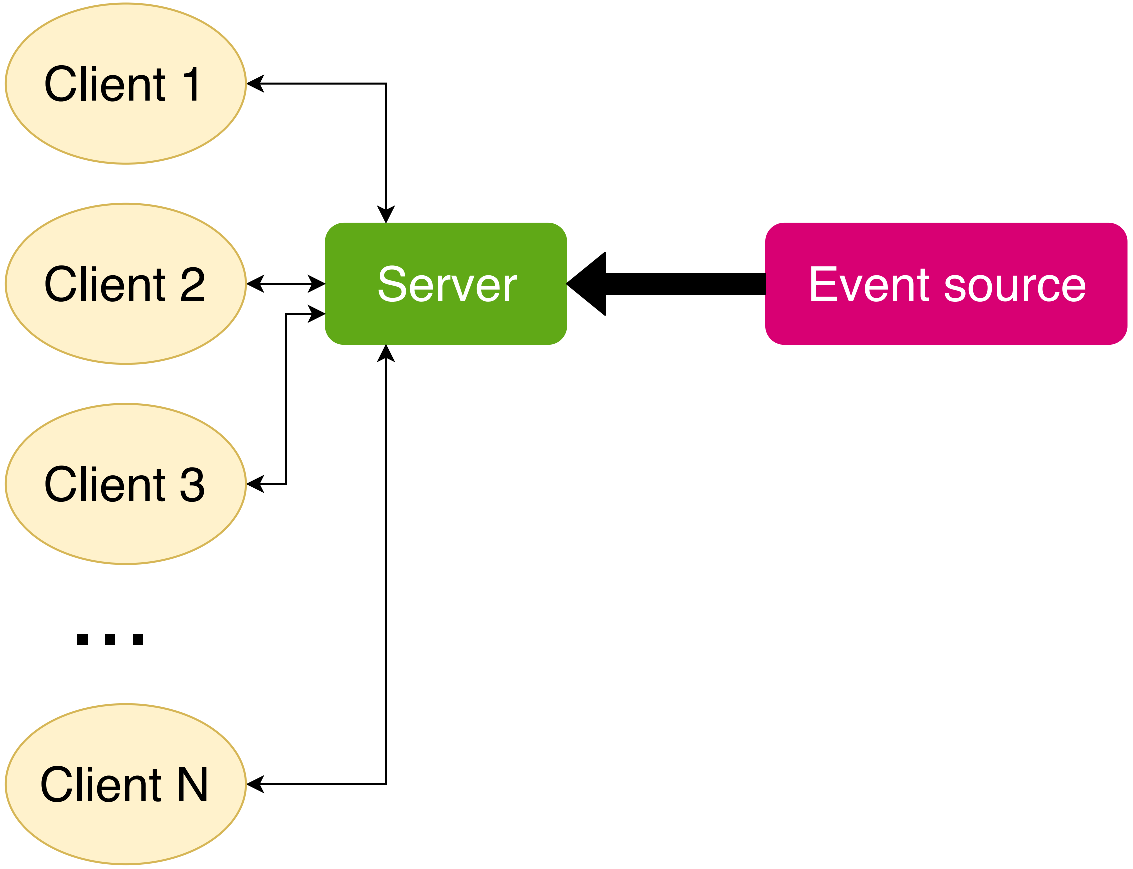
Performance

```scala
private[this] val list = 1 to 100

@Benchmark
def monixMerge: Int = {
  val observables = list
    .map(_ => Observable.fromIterable(list).executeAsync)

  Observable
    .merge(observables: _*)(OverflowStrategy.BackPressure(10))
    .foldL
    .runSyncUnsafe(1.seconds)
}

@Benchmark
def akkaMerge: Int = {
  val source: Source[Int, NotUsed] = Source(list)
  val f = list
    .map(_ => source)
    .fold(Source.empty)(_.merge(_))
    .runWith(Sink.fold(0)(_ + _))

  Await.result(f, 1.second)
}
```

Performance

```
# Run complete. Total time: 00:06:45
Do not assume the numbers tell you what you want them to tell.
Benchmark                         Mode  Cnt    Score    Error  Units
MonixBenchmark.akkaMerge         thrpt   10   46.207 ±  0.849  ops/s
MonixBenchmark.monixMerge        thrpt   10  531.182 ± 37.332  ops/s
```

Example



Example

```scala
val acceptClient: Task[(Long, Data)] = ???

def handleClientJoin(id: Long, data: Data, state: State): Task[State] = ???

def clientSubscriber(mState: MVar[State]) =
  Observable.repeat(())
    .doOnSubscribe(() => println(s"Client subscriber started"))
    .mapTask(_ => acceptClient)
    .mapTask { case (id, s) =>
      for {
        state <- mState.take
        newState <- handleClientJoin(id, s, state)
        _ <- mState.put(newState)
      } yield ()
    }
    .completedL
```

Example

```scala
val acceptEventSource: Task[Iterator[Event]] = ???

def handleEvent(event: Event, state: State): Task[State]

def eventSourceProcessor(mState: MVar[State]) =
  Observable.repeat(())
    .doOnSubscribe(() => println(s"Event processor started"))
    .mapTask(_ => acceptEventSource)
    .flatMap(it => Observable.fromIterator(it)
      .mapTask(e => for {
        state <- mState.take
        newState <- handleEvent(e, state)
        _ <- mState.put(newState)
      } yield ()))
    .headL
```

Example

```
for {
  initialState <- MVar(State())
  c = clientSubscriber(initialState).executeOn(clientScheduler)
  e = eventSourceProcessor(initialState).executeOn(eventSourceScheduler)
  _ <- Task.gatherUnordered(Seq(c, e))
} yield ()
```

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!