

Beyond the Future with purely functional Scala

Ilya Murzinov

 [ilya-murzinov](https://github.com/ilya-murzinov)

 [ilyamurzinov](https://t.me/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

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

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

Referential transparency

```
val s: Seq[String] = ???  
val v1 = transform(s) ++ transform(s)  
val wtfs = transform(s)  
val v2 = wtfs + wtfs
```

Referential transparency

```
val s: Seq[String] = ???  
val v1 = transform(s) ++ transform(s)  
val wtfs = transform(s)  
val v2 = wtfs + wtfs
```

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

Referential transparency

```
def load(wtfs: Seq[WTF]): Int = ???  
val s: Seq[WTF] = ???  
  
val v3 = load(s) + load(s)  
  
val result = load(s)  
val v4 = result + result
```

Referential transparency

```
def load(wtfs: Seq[WTF]): Int = ???  
val s: Seq[WTF] = ???  
  
val v3 = load(s) + load(s)  
  
val result = load(s)  
val v4 = result + result
```

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


Can we do better?

Yes we can!

Future FTW

```
import scala.concurrent.Future

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

Future FTW

```
import scala.concurrent.Future

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

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

Future FTW

```
import scala.concurrent.Future

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

```
val etlF: Future[Unit] = for {
  strings <- extractF
  wtfs = transform(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: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] two errors found  
[error] (Compile / compileIncremental) Compilation failed
```

ExecutionContext

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

ExecutionContext

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

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


ExecutionContext

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

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

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

ExecutionContext

```
val transformF: Seq[String] => Future[Seq[WTF]]

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

ExecutionContext

```
val transformF: Seq[String] => Future[Seq[WTF]]

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

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

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

ExecutionContext

```
val transformF: Seq[String] => Future[Seq[WTF]]

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

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

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

```
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 asynchronous

Drawbacks of Future

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

Drawbacks of Future

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

Can we do even better?

Yes we can!



Monix

Task[A]

Benefits of Task

- Lazy (ref. transparent)

Benefits of Task

- Lazy (ref. transparent)
- Cancellable

Benefits of Task

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

Benefits of Task

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

Benefits of Task

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

Benefits of Task

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

Benefits of Task

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

Scheduler

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

```
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

```
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
  .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
```

Thread shifting

```
> sbt runMain demo.Main
```

```
Running on thread: main
```

```
Running on thread: main
```

```
Running on thread: scala-execution-context-global-11
```

```
Running on thread: my-io-12
```

```
Ends on thread: scala-execution-context-global-11
```

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.execute0n(io)
  wtfs <- transformT(strings).execute0n(comp)
  _ <- loadT(wtfs).execute0n(io)
} yield ()
```

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

```
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: CancellableFuture[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

```
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

```
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

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

```
var effect = 0  
  
val source = Task.eval {  
    effect += 1  
    if (effect < 3) throw new RuntimeException("dummy") else effect  
}  
  
val cached = source.memoizeOnSuccess
```

Typical ETL

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

Observable[A]

Observable[A]

```
val extract0: Observable[Seq[String]] = ???  
val transformT: Seq[String] => Task[Seq[WTF]] = ???  
val loadT: Seq[WTF] => Task[Unit] = ???
```

Observable[A]

```
val extract0: Observable[Seq[String]] = ???  
val transformT: Seq[String] => Task[Seq[WTF]] = ???  
val loadT: Seq[WTF] => Task[Unit] = ???
```

```
val etl0: Task[Unit] =  
  extract0  
    .mapEval(transformT)  
    .mapEval(loadT)  
    .completedL
```

References

- Monix (<https://monix.io>)
- Alex Nedelcu's blog (<https://alexn.org/>)
- Scalaz 8 IO vs Akka (typed) actors vs Monix @ SoftwareMill

Questions?

Thanks!