Unveiling ZIO Test

A New Testing Library For Functional Scala

Agenda

- **Effects As Second Class Citizens**
- Effects As First Class Values
- Introducing ZIO Test

The Problem

Why do we need another testing framework?

Your testing framework needs to help you solve the "hard" problems:

- Concurrency
- Indeterminism
- Dependencies on other services
- Resource usage
- Multiple version and platforms

Back To The Future

Existing test frameworks use Future as their "effect" type.

We know Future has problems:
- Not actually a functional effect
- No support for interruption
- Not resource safe

How does this impact our ability to write tests?

Concurrency

ScalaTest

```scala
val forever: CatsIO[Int] = CatsIO
  .delay(println("Still going..."))
  .foreverM
  .as(1)

// Default timeout of 150 milliseconds
test("effects can be safely interrupted") {
  assert(forever.unsafeToFuture === Future.successful(1))
}
```

Specs2

```
val forever: CatsIO[Int] = CatsIO
  .delay(println("Still going..."))
  .foreverM
  .as(1)

def runawayTest =
  forever.unsafeToFuture must
    be_===(1).awaitFor(timeout = 1.second)
```

Indeterminism

ScalaTest

```scala
val random: CatsIO[Boolean] = CatsIO
  .delay(scala.util.Random.nextInt(100))
  .map(_ > 0)

test("random value is always greater than zero") {
  assert(random.unsafeToFuture === Future.successful(true))
}
```

Specs2

```
val random: CatsIO[Boolean] = CatsIO
  .delay(scala.util.Random.nextInt(100))
  .map(_ > 0)

def flakyTest =
  random.unsafeToFuture must beTrue.await
```

Dependencies On Other Services

ScalaTest

```scala
def sayHello(name: String): CatsIO[Unit] =
  CatsIO.delay(println(s"Hello, $name!"))

test("sayHello prints greeting with specified name") {
  assert(sayHello("Adam").unsafeToFuture === ???)
}
```

Resource Usage

Specs2

```
val kafkaResource: CatsResource[Kafka] =
  Resource.make(Kafka.acquire)(_.release)

class Specs2Spec extends Specification with BeforeAfterAll {
  val kafka = ???
  override def beforell = ???
  override def afterAll = ???
}
```

Multiple Versions And Platforms

ScalaTest

```scala
// JVM implementation
val javaVMName: CatsIO[Option[String]] =
  CatsIO.delay(Some(java.lang.System.getenv("java.vm.name")))

// ScalaJS implementation
val javaVMName: CatsIO[Option[String]] =
  CatsIO.pure(None)

test("Java virtual machine name can be accessed") {
  assert(javaVMName.unsafeToFuture === ???)
}
```

Specs2

```
// True on Scala 2, false on Dotty
val a: Double = Double.NaN
val eval = (a <= 0) || (10L <= 0)

def versionSpecific =
  (eval must be_===(???))
```

Agenda

- Effects As Second Class Citizens
- **Effects As First Class Values**
- Introducing ZIO Test

Been There, Done That

Modern functional effect systems have already solved most of these problems:

- Referential Transparency
- Interruptibility
- Resource Safety
- Environment Type

Tests As Values

```
type Test[+E, +S] = IO[TestFailure[E], TestSuccess[S]]
```

An effectual test that can fail with a E or succeed with an S.

Timing Out A Test

```
test.timeout(1.second)
```

Checking That A Test Is Not Flaky

```
test.repeat(Schedule.recurs(100))
```

Using A Resource

```
managed.use(testWithResource)
```

Agenda

- Effects As Second Class Citizens
- Effects As First Class Values
- **Introducing ZIO Test**

Taking The Idea To Its Logical Conclusion

```
type ZTest[-R, +E, +S] = ZIO[R, TestFailure[E], TestSuccess[S]]
```

An effectual test that requires an environment R and can fail with an E or succeed with an S.

Specs

```
type ZSpec[-R, +E, +L, +S]
```

Either a test or a suite containing other specs, annotated with labels of type L.

Test Aspects

```scala
trait TestAspect[+R0, -R1, +E0, -E1, +S0, -S1] {
  def apply[R >: R0 <: R1, E >: E0 <: E1, S >: S0 <: S1](
    spec: ZSpec[R, E, L, S]
  ): ZSpec[R, E, L, S]
}
```

A polymorphic function capable of transforming tests.

Concurrency

ZIO Test

```
testM("effects can be safely interrupted") {
  for {
    _ <- ZIO.effectTotal(println("Still going...")).forever
  } yield assertCompletes
} @@ timeout(1.second)
```

Indeterminism

ZIO Test

```
testM("random value is always greater than zero") {
  assertM(random.nextInt(100), isGreaterThan(0))
} @@ nonFlaky,
```

Dependencies On Other Services

ZIO Test

```
def sayHello(name: String): URIO[Console, Unit] =
  console.putStrLn(s"Hello, $name!")

testM("sayHello prints greeting with specified name") {
  for {
    _ <- sayHello("Adam")
    output <- TestConsole.output
  } yield assert(output, equalTo(Vector("Hello, Adam!\n")))
}
```

Resource Usage

ZIO Test

```scala
val kafka: Managed[Nothing, Kafka] =
  Managed.make(Kafka.acquire)(_.release)

suite("shared resources can be provided")(
  testM("first Kafka test") {
    assertM(ZIO.accessM[Kafka](_.use), anything)
  },
  testM("second Kafka test") {
    assertM(ZIO.accessM[Kafka](_.use), anything)
  }
).provideManagedShared(kafka)
```

Multiple Versions And Platforms

ZIO Test

```
testM("Java virtual machine name can be accessed") {
  assertM(
    live(system.property("java.vm.name")),
    isSome(containsString("VM"))
  )
} @@ jvmOnly
```

Composability

ZIO Test

Run a test and check that it is not flaky, but only on the JVM. Timeout the test after 60 seconds no matter what.

ZIO Test

Run a test and check that it is not flaky, but only on the JVM. Timeout the test after 60 seconds no matter what.

```
test @@ jvm(nonFlaky) @@ timeout(60.seconds)
```

Property Based Testing

Unified Property Based Testing

We don't need a separate library for property based testing, ZIO's effect type has all the power we need:

- State
- Random Number Generation

Gen

```
final case class Gen[-R, +A](
  sample: ZStream[R, Nothing, Sample[R, A]]
)
```

A generator represents a generator of values of type A, which requires an environment R.

Gen

```
final case class Gen[-R, +A](
  sample: ZStream[R, Nothing, Sample[R, A]]
)
```

A random generator is an effectual stream with a single element.

A deterministic generator is a stream with multiple elements.

Sample

```scala
final case class Sample[-R, +A](
  value: A,
  shrink: ZStream[R, Nothing, Sample[R, A]]
)
```

A sample is a value along with a tree of potential "shrinkings" of that value.

ScalaCheck

```
val positiveInts = Gen.nonEmptyListOf(Gen.choose(1, 10))

property("product of positive integers is greater than sum") =
  forAll(positiveInts) { ints =>
    ints.product >= ints.sum
  }
```

ScalaCheck

```scala
val positiveInts = Gen.nonEmptyListOf(Gen.choose(1, 10))

property("product of positive integers is greater than sum") =
  forAll(positiveInts) { ints =>
    ints.product >= ints.sum
  }

// Property failed with counterexample: List(0, 1)
// But 0 is not a positive integer!
```

ZIO Test

```
val positiveInts = Gen.listOf1(Gen.int(1, 10))

testM("product of positive integers is greater than sum") {
  check(positiveInts) { ints =>
    assert(ints.product, isGreaterThanEqualTo(ints.sum))
  }
}
```

ZIO Test

```scala
val positiveInts = Gen.listOf1(Gen.int(1, 10))

testM("product of positive integers is greater than sum") {
  check(positiveInts) { ints =>
    assert(ints.product, isGreaterThanEqualTo(ints.sum))
  }
}

// Property failed with counterexample: List(1, 1)
```

Assertions

Assertion

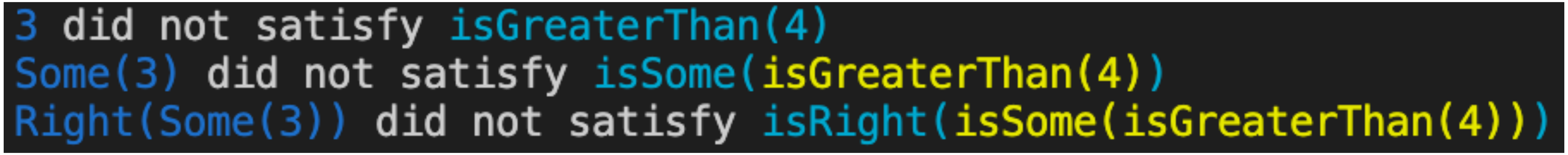
```
class Assertion[-A] extends (A => AssertResult)
```

An assertion is capable of producing assertion results on an A.

ZIO Test

```
test("value is in specified range") {
  assert(
    Right(Some(3)),
    isRight(isSome(isGreaterThan(4)))
  )
},
```

ZIO Test



```
3 did not satisfy isGreaterThan(4)
Some(3) did not satisfy isSome(isGreaterThan(4))
Right(Some(3)) did not satisfy isRight(isSome(isGreaterThan(4)))
```

Conclusion

- Testing frameworks need to evolve
- Available to users of other effect types through interop package
- Join us on Discord and Github

Thank You!