Scala Ecosystem: Cats Effect

Mikhail Mutcianko, Alexey Shcherbakov

СПБгУ, СП

22 апреля 2021

Effect

In functional programming an *effect* is a context in which your computation operates:

- `Option` models the absence of value
- `Try` models the possibility of failure
- `Future` models the asynchronicity of a computation
- ...

Side effects thereby denote computational leaks that escape your effect, or context

Referential Transparency

An expression is said to be *referential transparent* when it can be replaced by its value without changing the program

```
def sum(a: Int, b: Int): Int = a + b

assertEquals(5, sum(2,3))
```

Principle

How to make code that performs side-effects referentially transparent?

Principle

How to make code that performs side-effects referentially transparent?

Not to run it

Principle

How to make code that performs side-effects referentially transparent?

Not to run it

`cats.effect.IO` does exactly this: it wraps a computation but doesn't run it

```scala
val n = IO {  // doesn't print anything
  print("n")
  3
}
```

IO basics

You run an `IO` by calling one of the “unsafe” method that it provides:

- `unsafeRunSync` runs the program synchronously
- `unsafeRunTimed` runs the program synchronously but abort after a specified timeout
- `unsafeRunAsync` runs the program asynchronously and execute the specified callback when done
- `unsafeToFuture` runs the program and produces the result as a Scala Future
- ...

Conditional running

By not running the code straight away we get additional capabilities. E.g. it's possible to decide to run a computation after declaring it:

```scala
val isWeekday = true
for {
    _ <- IO(println("Working")).whenA(isWeekday)
    _ <- IO(println("Offwork")).unlessA(isWeekday)
} yield ()
```

Async IO

When you need to interact with remote systems and in this case you need an asynchronous IO. An asynchronous IO is created by passing a callback that is invoked when the computation completes.

```
def fromCompletableFuture[A](f: => CompetableFuture[A]): IO[A] =
  IO.async { callback =>
    f.whenComplete { (res: A, error: Throwable) =>
      if (error == null) callback(Right(res))
      else callback(Left(error))
    }
  }
```

Brackets

IO being lazy it's possible to make sure all the resources used in the computation are released properly no matter the outcome of the computation. Think of it as a `try / catch / finally`. In cats-effect this is called `Bracket`

```
// acquire the resource
IO(new Socket("hostname", 12345)).bracket { socket =>
  // use the socket here
} { socket =>
  // release block
  socket.close()
}
```

Resources

Resource builds on top of Bracket. It allows to acquire resources (and making sure they are released properly) before running your computation. Moreover Resource is composable so that you can acquire all your resources at once.

```scala
def acquire(s: String) = IO(println(s"Acquire $s")) *> IO.pure(s)
def release(s: String) = IO(println(s"Releasing $s"))
val resources = for {
   a <- Resource.make(acquire("A"))(release("A"))
   b <- Resource.make(acquire("B"))(release("B"))
} yield (a ,b)

resources.use { case (a, b) => // use a and b
} // release code is automatically invoked when computation finishes
```

Cancellation

Async `IO` takes `Callback => Unit`

To be able to cancel, we need to somehow pass another `IO` that's able to cancel the computation

```scala
def cancelable[A](k: (Either[Throwable, A] => Unit) => IO[Unit]): IO[A] = ???
def fromCompletableFuture[A](f: => CompletableFuture[A]): IO[A] =
  IO.cancellable { callback =>
    f.whenComplete(res: A, error: Throwable) =>
      if (error == null) callback(Right(res))
      else               callback(Left(error))
    IO(f.cancel(true)) // cancelling IO
  }
```