## Poor Man's Concurrency Monad - Scala

Bookmark this page

# Poor Man's Concurrency Monad - Scala

In this lab we are going to implement a Monad for handling concurrency. You must use the provided template for this lab.

We are going to simulate concurrent processes by interleaving them. Interleaving implements concurrency by running the first part of one process, suspending it, and then allowing another process to run, etcetera, etcetera.

A process is represented by the following abstract class `Action` that encodes primitive actions that perform a side-effect and then return a continuation action, or the concurrent execution of two actions, or an action that has terminated.

```
abstract sealed class Action
case class Atom(atom: Unit => Action) extends Action {
    override def toString() = "atom"
}
case class Fork(a1: Action, a2: Action) extends Action {
    override def toString = s"fork ${a1 toString} ${a2 toString}"
}
case class Stop() extends Action {
    override def toString = "stop"
}
```

To suspend a process, we need to grab its "future" and store it away for later use. Continuations are an excellent way of implementing this. We can change a function into continuation passing style by adding an extra parameter, the continuation, that represents the "future" work that needs to be done after this function terminates. Instead of producing its result directly, the function will now apply the continuation to the result.

Given a computation of type `Action`, a function that uses a continuation with result type `A` has the following type:

```
(A => Action) => Action
```

This type can be read as a function that takes as input a continuation function `(A => Action)`, that specifies how to continue once the result of type `A` of the current computation is available. An application `f(c)` of this type will call `c` with its result when it becomes available.

Because we want to make `((A => Action) => Action)` into a monad, we first need to wrap into a trivial algebraic data type.

```
class Concurrent[A](val func: (A => Action) => Action) {
    // some functions
}
```

As we will emphasize several times in the exercises below, we find it easiest to derive the implementations by ignoring the wrapper (think like a fundamentalist) since that makes "listening to the types" easier, and then add the boilerplate and constructor calls to make Scala happy (code like a hacker). This may, or may not, hold for you.

## Exercise 0

To express the connection between an expression of type `Concurrent[A]` and one of type `Action`, we define a function `def action(): Action` in the `Concurrent[A]` class that transforms a `((A => Action) => Action)` into an `Action` that uses `case class Stop() extends Action` to create the continuation to the `Concurrent[A]`.

As always, let the types guide you. There is only one obvious way to create value of type `a => Action` from the value `case class Stop() extends Action`. Then when you get a value of type `((a => Action) => Action)` there is only one way to combine these two to obtain a value of type `Action`.

Implement the function:

```
def action(): Action
```

## Exercise 1

To make the subtypes of `Action` easily accessible, we can define helper functions that hide the boilerplate required to use them.

The first helper function that we will define is the function `def stop[A](): Concurrent[A]`, which *discards* any continuation, thus ending a computation.

Thus we need to return a function of type `((Action => ()) => ())` wrapped in the `Concurrent` class. THis function takes a continuation, which gets discarded, and then it returns a `Stop` action.

Implement the helper function:

```
def stop[A](): Concurrent[A]
```

## Exercise 2

Now we can define the helper function `def atom[A](ioA: Unit => A): Concurrent[A]`, which turns an arbitrary computation in Haskell's `IO` Monad into an atomic action represented using the `Atom` constructor. As you already know, `IO` monad is used in Haskell to perform side effects. Since Scala allows side effects in the code, we don't really need this `IO` monad. However, since we want to emulate Haskell's lazy evaluation, we emulate the `IO` monad by using a function of type `Unit => A`.

The easiest way to implement this function is to first implement `def atom[A](ioA: Unit => A): (A => Action) => Action` by taking a value `ioA: Unit => A` and returning a value of type `(A => Action) => Action` which looks like `(c: a => Action) => ... value of type Action ...`

Now the only step that is left is to wrap and unwrap the `Concurrent` class, to implement `def atom[A](ioA: Unit => A): Concurrent[A]`.

Implement the function:

```
def atom[A](ioA: Unit => A): Concurrent[A]
```

## Exercise 3

In order to access `Fork`, we need to define two operations. The first, called `def fork(): Concurrent[Unit]`, is defined in the `Concurrent[A]` class, which forks the `val func: (A => Action) => Action` by turning it into an action and continues by passing `()` as the input to the continuation.

The second, `def par[A](c1: Concurrent[A], c2: Concurrent[A]): Concurrent[A]`, combines two computations into one by forking them both and passing the given continuation to both parts.

Implement the functions:

```
def fork(): Concurrent[Unit]
```

```
def par[A](c1: Concurrent[A], c2: Concurrent[A]): Concurrent[A]
```

## Exercise 4

To make `Concurrent` a `Monad`, we need to provide implementations of Haskell's `(>>=)` and `return`. In Scala these functions are called `flatMap` and `of`. Since the staff is nice, we will give you `of` for free. But you will have to define `flatMap` yourself.

```
def flatMap[B](mapper: A => Concurrent[B]): Concurrent[B] = ???
```

```
def of[A](a: A) = new Concurrent((cont: A => Action) => cont(a))
```

Don't panic! Let the types guide you and everything will be alright. There is really just one way to wire up all the pieces you have on the table to create a result of the required type. Don't try to understand what the code does operationally, trust the types.

The easiest way to do this is by first developing on a **piece of scratch paper** (*), a function `def flatMap[B](f: (A => Action) => Action,  g: A => ((B => Action) => Action)): (B => Action) => Action` ignoring the `Concurrent` wrapper. Now you can let the types lead you to the only reasonable implementation. Once you've found this, you can add the boilerplate and applications of `Concurrent`. Your implementation without `Concurrent` might look as follows in Haskell notation:

```
ma >>= f = .... given ma::((a -> Action) -> Action) .....

           .... and f::(a -> ((b -> Action) -> Action)) ....

           .... create a result of type ((b -> Action) -> Action) ....
```

Remember when you return a value of a function type, such as `((B => Action) => Action)`, the value you create looks like `c => ... expression of type Action ....`

Similarly, when you need to pass a value of a function type, for example `A => ((B => Action) => Action)`, the value you pass an expression of the form: `a => ... expression of type ((B => Action) => Action) ....`

In the end, the solution only needs two lambda expressions and a couple of function applications.

While doing all this, you don't need to look at the structure of `Action` at all. In fact, this would work for any type instead of `Action`.

(*) You can use Scala Worksheets as your scratch paper. This is what we mean by "*Think like a Fundamentalist*". Implementing `flatMap` with all the nasty wrapping is "*Code like a Hacker*". Thinking like a fundamentalist is like Dolby noise reduction for the Hacker.

Implement `flatMap` for the `Concurrent` monad.

## Exercise 5

At any moment, the status of the computation is going to be modelled by a list of "concurrently running" actions. We will use a scheduling technique called round robin to interleave the processes. The concept is easy: take the first process from the list, run its first part, take the continuation and put that at the back of the list. Keep doing this until the list is empty.

We implement this idea in the function `def roundRobin(list: List[Action]): () => Unit`. An `Atom` monadically executes its argument and puts the resulting process at the back of the process list. `Fork` creates two new processes and `Stop` discards its process. Make sure you leverage the helper functions you defined before.

Implement the function:

```
def roundRobin(list: List[Action]): () => Unit
```

## LET'S FINISH

To finish this lab, go to the question page and answer all 26 of them.

English

POWERED BY OPENedX

Download on the App Store

ANDROID APP ON Google play