Poor Man's Concurrency Monad - Haskell

Bookmark this page

Poor Man's Concurrency Monad - Haskell

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, et cetera, et cetera.

A process is represented by the following recursive algebraic data type `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.

```
data Action = Atom (IO Action)
            | Fork Action Action
            | 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.

Unfortunately, because we want to make `(a -> Action) -> Action` into a monad, we first need to wrap it into a trivial algebraic data type, which we have to wrap and unwrap when implementing the monad operators. You might remember that in the exercises on the `Parser` monad we used a `newtype` declaration but since we are ignoring bottoms the difference is insignificant.

```
data Concurrent a = Concurrent ((a -> Action) -> Action)
```

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 pattern matching and constructor calls to make Hugs 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 `action :: Concurrent a -> Action` that transforms a `((a -> Action) -> Action)` into an `Action` that uses `Stop :: Action` to create the continuation to the `Concurrent a` passed as the first argument to `action`.

The easiest road to implement this function is to initially ignore the `Concurrent` wrapper, and first define a function `action :: ((a -> Action) -> Action) -> Action` and later add the pattern-matching to remove the wrapper and transform a value of type `Concurrent a` into a value of type `((a -> Action) -> Action) -> Action`. As always, let the types guide you. There is only one obvious way to create a value of type `a -> Action` from the value `Stop :: Action`. Then when you get a value of type `ma :: ((a -> Action) -> Action)` there is only one way to combine these two to obtain a value of type `Action`.

Implement the function `action`:

```
action :: Concurrent a -> Action
```

Exercise 1

To make the constructors of the data type `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 `stop :: Concurrent a`, which *discards* any continuation, thus ending a computation.

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

Implement the helper function *stop*:

```
stop :: Concurrent a
```

Exercise 2

Now we can define the helper function `atom :: IO a -> Concurrent a`, which turns an arbitrary computation in the `IO` Monad into an atomic action represented using the `Atom` constructor.

The easiest way to implement this function is to first implement `atom :: IO a -> ((a -> Action) -> Action)` by taking an value `x :: IO a` and returning a value of type `((a -> Action) -> Action)` which looks like `\c :: (a -> Action) -> ... value of type Action ...`

You already know, from the previous homework and labs, how to combine a value of type `IO a` and a function of type `a -> IO b` into a value of type `IO b` using `(>>=)`, in this case `b` is instantiated to `Action`. You also know how to convert a value of type `Action` into a value of type `IO Action` using `return`. Finally, the obvious choice to turn a value of type `IO Action` into an `Action` is by using the `Atom` constructor. With all these pieces on the table, there is really only one way to wire them together to implement your function. Now the only step that is left is to wrap and unwrap the `Concurrent` data type, to implement `atom :: IO a -> Concurrent a`.

Implement the function `atom`:

```
atom :: IO a -> Concurrent a
```

Exercise 3

In order to access `Fork`, we need to define two operations. The first, called `fork :: Concurrent a -> Concurrent ()`, forks its argument by turning it into an action and continues by passing `()` as the input to the continuation:

```
fork :: Concurrent a -> Concurrent ()
```

The second, `par :: Concurrent a -> Concurrent a -> Concurrent a`, combines two computations into one by forking them both and passing the given continuation to both parts.

```
par :: Concurrent a -> Concurrent a -> Concurrent a
```

Implement the functions `fork` and `par`.

Exercise 4

To make `Concurrent` an instance of the `Monad` type class, we need to provide implementations of `(>>=)` and `return`. Since the staff is nice, we will give you `return` for free. But you will have to define `(>>=)` yourself.

```
instance Monad Concurrent where
    (Concurrent f) >>= g = ...
    return x = Concurrent (\c -> c x)
```

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 `(>>=) :: ((a -> Action) -> Action) -> (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 pattern matching and applications of `Concurrent` that Haskell unfortunately requires. Your implementation without `Concurrent` will look as follows:

```
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 can be 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 Hugs as your scratch paper by using a different name than `(>>=)`, for example `bind`. This is what we mean by "*Think like a Fundamentalist*". Implementing `(>>=)` with all the nasty wrapping is "*Code like a Hacker*". Thinking like a fundamentalist is like Dolby noise reduction for the Hacker.

Implement `(>>=)` for the `Concurrency` monad.

Exercise 5

At any moment, the status of the computation is going to be modelled as 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, then take the resulting 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 `roundRobin :: [Action] -> IO ()`. 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 `roundRobin`:

```
roundRobin :: [Action] -> IO ()
```

LET'S FINISH

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