Overview of free monad in cats

Cats (http://typelevel.org/cats/) library, which helps to write purely functional code, is a quite young project in Scala. It brings many structures and functional constructs for you to use. The library itself is designed in a modular way, so we can use only things that we really need. In this post I'll try to look at the Free Monad concept and implement basic functionality of LOGO (http://www.calormen.com/jslogo/) programming language.

What is a Free Monad?

Basically, you can look at Free Monad as a way to describe your program. When I say program I mean set of abstract instructions. More precisely, instructions are some kind of AST (Abstract Syntax Tree) that represents computations. The Free Monad allows to combine them into functions, that later can be combined into higher level functions, which should improve composability of our code. Moreover, the usage of free monad removes the necessity of tinkering with implementation details. Furthermore, API of our code isn't strictly defined as `Option`, `Future` or other monads.

Such a description then must be interpreted, which basically means that we're going to run it using an interpreter with an implementation of our AST. In the interpreter we start to worry about implementation details. But these details are separated from our business logic, what makes it easier to test. Moreover, the same function can be executed in different contexts, even using various interpreters. That makes this pattern very useful.

The monad `M[A]` has two important functions:

```
1  def pure[A](a: A): M[A]
2  def flatMap[A, B](fa: M[A])(f: A => M[B]): M[B]
```


where `A` has to be a type constructor(`A[_]`), which means that type `A` needs to be parameterized by some other type. Using the `flatMap` it's possible to write our program flow control, working only on a defined instruction set.

What does the Free Monad look like in Cats? It's defined as follows:

```
1  sealed abstract class Free[S[_], A]
```


But there is a convenient method for lifting `S[A]` into `Free[S[_], A]` in the companion object `Free`.

```
1  def liftF[F[_], A](value: F[A]): Free[F, A]
```


Usage of this method looks like that:

```
1  trait Container[T]
2  case class StringContainer(value: String) extends Container[String]
3  val lifted: Free[Container, String] = Free.liftF[Container, String](StringContainer("foo"))
```


We'll see how it works in practice.

Putting the Free monad to work

We gonna implement basic instructions from LOGO (http://www.calormen.com/jslogo/) language - the famous “Turtle”. We will implement movement in a 2 dimensional space.

Creating set of instructions (AST)

Ok, let’s say we have some set of instructions that we want to use in order to create program. Using the LOGO example I would like to implement basic functionality like moving forward and backward, rotating left or right and showing the current position. To do that I’ve created a few simple classes to represent those actions.

```
object Logo {
  sealed trait Instruction[A]
  case class Forward(position: Position, length: Int) extends Instruction[Position]
  case class Backward(position: Position, length: Int) extends Instruction[Position]
  case class RotateLeft(position: Position, degree: Degree) extends Instruction[Position]
  case class RotateRight(position: Position, degree: Degree) extends Instruction[Position]
  case class ShowPosition(position: Position) extends Instruction[Unit]
}
```

post.scala **view raw (https://gist.github.com/anonymous/2aa14fd93f68e49494a830501c699ff0/raw/74932288c7abafdbbf40bd4504951ac92ba5390f/post.scala) (https://gist.github.com/anonymous/2aa14fd93f68e49494a830501c699ff0#file-post-scala)** hosted with ❤ by **GitHub (https://github.com)**

These are just simple case classes. Type `A` will be the type that Free Monad will be working on. That means if we use a `flatMap` we will have to provide a function of type `A => Instruction[B]`.

There are also two classes that will be used. It’s a part of simplified domain model.

```
case class Position(x: Double, y: Double, heading: Degree)
case class Degree(private val d: Int) {
  val value = d % 360
}
```

post.scala **view raw (https://gist.github.com/anonymous/dfc78d8211e021c50c12289e1ed543e5/raw/d0de4041efef1b183662081a4b3c1a127ff9b4c9/post.scala) (https://gist.github.com/anonymous/dfc78d8211e021c50c12289e1ed543e5#file-post-scala)** hosted with ❤ by **GitHub (https://github.com)**

At this point we have defined only our actions with some additional types, but it does not compute anything, it’s abstract.

Creating the DSL

DSL is a Domain Specific Language. It contains functions that refer to the domain. Such functions make the code more readable and expressive. To make it easier to use I’ve created some helper methods that will create free monads wrapping our actions, by lifting `Instruction[A]` into `Free[Instruction, A]`.

```
def forward(pos: Position, l: Int): Free[Instruction, Position] = Free.liftF(Forward(pos, l))
def backward(pos: Position, l: Int): Free[Instruction, Position] = Free.liftF(Backward(pos, l))
def left(pos: Position, degree: Degree): Free[Instruction, Position] = Free.liftF(RotateLeft(pos, degree))
def right(pos: Position, degree: Degree): Free[Instruction, Position] = Free.liftF(RotateRight(pos, degree))
def showPosition(pos: Position): Free[Instruction, Unit] = Free.liftF(ShowPosition(pos))
```

post.scala **view raw (https://gist.github.com/anonymous/0706d8143c2f33349abd43ed244e58b5/raw/ffb88740405576dbc86a4593132012485e27f964/post.scala) (https://gist.github.com/anonymous/0706d8143c2f33349abd43ed244e58b5#file-post-scala)** hosted with ❤ by **GitHub (https://github.com)**

These methods are used to write the application and make it stateless by taking a position as a parameter and not storing it inside the function.

The program

Now we can easily use our DSL. Having the methods in place we can use them in a for comprehension to build a program description. Let's say we want to get from point (0,0) to point (10,10). The code looks like this:

```
1  val program: (Position => Free[Instruction, Position] = {
2    start: Position =>
3       for {
4          p1 <- forward(start, 10)
5          p2 <- right(p1, Degree(90))
6          p3 <- forward(p2, 10)
7       } yield p3
8  }
```

post.scala **view raw (https://gist.github.com/anonymous/118bc52627d63aca9fa877a8397b895b/raw/8aa92e19c407f94e6327bb27790b8812d247bbea/post.scala) (https://gist.github.com/anonymous/118bc52627d63aca9fa877a8397b895b#file-post-scala)** hosted with ❤ by **GitHub (https://github.com)**

As we can see the program is just a function that takes a starting position and moves the turtle. The result of calling the function is also a Free monad and that's great news because it's very easy to compose another function using this one. We can simply think of creating functions to draw basic geometric shapes and then using these functions to draw something more complicated; furthermore, it's going to compose very well. It's like that because of referential transparency property of the free monad.

Another advantage is that we don't worry how it is computed. We are just expressing how program should work using the DSL.

Interpreter

At this moment we have our program description, but we want to execute it. In order to do that a `NaturalTransformation` is needed. It's a function that can transform one type constructor into another one. A `NaturalTransformation` from `F` to `G` is often written as `F[_] ~> G[_]`. What is important, is that we have to provide implicit conversion from `G` to a `Monad[G]`. It's possible to have multiple interpreters, e.g. one for testing and another one for production. This is very useful when we create an AST for interacting with a database or some other external service.

The simplest monad is the identity monad `Id`, which is defined in cats as a simple type container.

```
1  type Id[A] = A
```

post.scala **view raw (https://gist.github.com/anonymous/399358fce3cbe2193c8e193b06c49343/raw/f71c15e0298a886ff8e9bd4327eb4ff19612a9cc/post.scala) (https://gist.github.com/anonymous/399358fce3cbe2193c8e193b06c49343#file-post-scala)** hosted with ❤ by **GitHub (https://github.com)**

It has all necessary functions implemented and implicit value that is a `Monad[Id]`. In the `cats` package object there is an implicit instance that helps with conversion to an `Id`. Let's use it to write our own interpreter.

```
 1  object InterpreterId extends (Instruction ~> Id) {
 2    import Computations._
 3    override def apply[A](fa: Instruction[A]): Id[A] = fa match {
 4      case Forward(p, length) => forward(p, length)
 5      case Backward(p, length) => backward(p, length)
 6      case RotateLeft(p, degree) => left(p, degree)
 7      case RotateRight(p, degree) => right(p, degree)
 8      case ShowPosition(p) => println(s"showing position $p")
 9    }
10  }
```

post.scala **view raw (https://gist.github.com/anonymous/d3a5bd03187d7cd786ed0a5270ef5c8e/raw/18af2e2441611b513607119210576f6af1e70a45/post.scala) (https://gist.github.com/anonymous/d3a5bd03187d7cd786ed0a5270ef5c8e#file-post-scala)** hosted with ❤ by **GitHub (https://github.com)**

`InterpreterId` is defined as a natural transformation from `Instruction` to `Id`. `Computations` object has all the functions that are necessary to compute a new position. Functions used in first 4 cases return value of type `Position` which is equal to `Id[Position]`. The `Id` does not modify value, it is just a container that provides monadic functions. If you put `Position` value `pos` into `Id[Position].pure(pos)` it will return value `pos` of type `Position`. If you want to `map` over the `Id` using function `f: Position => B`, it will behave the same as if you apply the `pos` into the `f`.

Running the program

To run the `program` we need to simply `foldMap` it using the interpreter and pass a parameter to the function to start evaluation.

```
1  val startPosition = Position(0.0, 0.0, Degree(0))
2
3  program(startPosition).foldMap(InterpreterId)
```

post.scala **view raw (https://gist.github.com/anonymous/dcb3e9d0cad990cd93938a9e6a876da8/raw/26bc5fde3c399ef667da3f0e935ea2f8ea55699b/post.scala) (https://gist.github.com/anonymous/dcb3e9d0cad990cd93938a9e6a876da8#file-post-scala)** hosted with ❤ by **GitHub (https://github.com)**

The result of this operation will be just a `Position` because last the instruction of `program` was `forward` and we yielded the result of it. When we look at the definition of `Forward` we can see that it extends `Instruction[Position]` and that type parameter specifies that we will have a value of `Position` type as result.

Another Interpreter

Let's assume we want to move on a surface that has only non-negative coordinates. Whenever the value of `x` or `y` in `Position` becomes negative we want to stop further computation. The simplest solution is to change the interpreter so that it'll transform `Instruction` into `Option`. Function `foldMap` is using the `flatMap` function of the `Monad` that our `Instruction` is transformed into. So if we going to return `None` then the computation will be stopped. Let's implement it.

```
 1  object InterpretOpt extends (Instruction ~> Option) {
 2    import Computations._
 3    val nonNegative: (Position) => Option[Position] = {
 4      p => if (p.x >= 0 &&p.y >= 0) Some(p) else None
 5    }
 6
 7    override def apply[A](fa: Instruction[A]) = fa match {
 8      case Forward(p, length) => nonNegative(forward(p, length))
 9      case Backward(p, length) => nonNegative(backward(p, length))
10      case RotateLeft(p, degree) => Some(left(p, degree))
11      case RotateRight(p, degree) => Some(right(p, degree))
12      case ShowPosition(p) => Some(println(s"showing position $p"))
13    }
14  }
```

post.scala **view raw (https://gist.github.com/anonymous/f90ca2bffd5461b25063fc88d43bb1cd/raw/7820bee17163fcb721a8c1469eac0de95822e1b2/post.scala) (https://gist.github.com/anonymous/f90ca2bffd5461b25063fc88d43bb1cd#file-post-scala)** hosted with ❤ by **GitHub (https://github.com)**

We defined `nonNegative` as a part of the interpreter - the reason is that it's an implementation detail not connected to the business logic. And now if we run the program with such definition

```
 1  val program2: (Position => Free[Instruction, Unit]) = {
 2      s: Position =>
 3        for {
 4          p1 <- forward(s, 10)
 5          p2 <- right(p1, Degree(90))
 6          p3 <- forward(p2, 10)
 7          p4 <- backward(p3, 20)//Here the computation stops, because result will be None
 8          _ <- showPosition(p4)
 9        } yield ()
10    }
```

post.scala **view raw (https://gist.github.com/anonymous/1c26dcd1cd74acd4c6d810a28f660422/raw/061595228278a11aca2b221689c7c839c65b96c9/post.scala) (https://gist.github.com/anonymous/1c26dcd1cd74acd4c6d810a28f660422#file-post-scala)** hosted with ❤ by **GitHub (https://github.com)**

It'll not print the position, so we achieved our goal. We can easily think about another interpreter that could be implemented for this AST. It could be a graphical representation of our program.

Composing

Free Monads are a really powerful tool. One of the reasons is composition. Our example is rather simple, but you can imagine that you've built a whole instruction set for your business logic. It is completely separated from other code in application. Then you can add set of instructions for logging, accessing the database, failure handling or just another business logic but for some reason separated from the previous one. Each of such DSLs can be easily tested.

Writing the program doesn't change much. You have to do some adjustments. Let's say we want to add to our LOGO application two new instructions - `PencilUp` and `PencilDown`, but they will be in other instruction set. First thing is defining `PencilInstruction`

```
1  sealed trait PencilInstruction[A]
2  case class PencilUp(position: Position) extends PencilInstruction[Unit]
3  case class PencilDown(position: Position) extends PencilInstruction[Unit]
```

post.scala **view raw (https://gist.github.com/anonymous/451c17650c677b2e2dfd04553bd84a4e/raw/76616e6bde425f0272a6a8d71b61425d607c8bc3/post.scala) (https://gist.github.com/anonymous/451c17650c677b2e2dfd04553bd84a4e#file-post-scala)** hosted with ❤ by **GitHub (https://github.com)**

The problem with the program type is that `PencilInstruction` and `Instruction` don't have a common supertype, and they shouldn't have. We need to define some type that will be either former or latter of these two type constructors and still be able to pass them type parameter. Luckily, there is a `Coproduct`, which is made exactly for such tasks. It's a wrapper for `Xor` type, which is more or less the same thing as Scala's `Either`. `Coproduct` requires two type constructors and type that will be inserted into them. In this way we can make superset of two sets of instructions and create a supertype for them.

```
1  final case class Coproduct[F[_], G[_], A](run: F[A] Xor G[A])
```

post.scala **view raw (https://gist.github.com/anonymous/32ff84bfab9eafb0dbb81bfcdf3d9fe7/raw/0bbd62c7119969d79051ca7207aedbe3fdc5fd84/post.scala) (https://gist.github.com/anonymous/32ff84bfab9eafb0dbb81bfcdf3d9fe7#file-post-scala)** hosted with ❤ by **GitHub (https://github.com)**

Let's define our common type.

```
1  type LogoApp[A] = Coproduct[Instruction, PencilInstruction, A]
```

post.scala **view raw (https://gist.github.com/anonymous/2ad6158c73ee2b0001a2ee0bfe2498cc/raw/505aa6ded113c52963317c81071dcc311a6e5adf/post.scala) (https://gist.github.com/anonymous/2ad6158c73ee2b0001a2ee0bfe2498cc#file-post-scala)** hosted with ❤ by **GitHub (https://github.com)**

In application we will be using `LogoApp` as a whole set of instructions. To make the mixing of these two ASTs possible we need to be able to lift both of them to `Coproduct` type. To do this we have to change our lifting methods - instead of using `Free.liftF` method we will use an injecting function.

```
1  final class FreeInjectPartiallyApplied[F[_], G[_]] private[free] {
2    def apply[A](fa: F[A])(implicit I : Inject[F, G]): Free[G, A] =
3      Free.liftF(I.inj(fa))
4    }
5
6  def inject[F[_], G[_]]: FreeInjectPartiallyApplied[F, G] = new FreeInjectPartiallyApplied
```

post.scala **view raw (https://gist.github.com/anonymous/4e81c4dd044caa3d36a735262fd497c2/raw/05e4af69900343cddea26b4f5d864b6c22a3e6a9/post.scala) (https://gist.github.com/anonymous/4e81c4dd044caa3d36a735262fd497c2#file-post-scala)** hosted with ❤ by **GitHub (https://github.com)**

It basically means that we can lift our `Instruction` or `PencilInstruction` set into the `Coproduct` which is the superset of both of them. Because we want to be flexible about the `Coproduct` types we will define classes wrapping our DSL. These classes will take type parameter, which will be corresponding to the `Coproduct`. This is what our Logo definition will look like

```
1  object Logo{
2
3    sealed trait Instruction[A]
4    ...
5    sealed trait PencilInstruction[A]
6    ...
7    case class Position(x: Double = 0, y: Double = 0, heading: Degree = Degree())
8    case class Degree(private val d: Int = 0) {...}
9
```

```
10    object dsl {
11      class Moves[F[_]](implicit I: Inject[Instruction, F]) {
12        def forward(pos: Position, l: Int): Free[F, Position] = Free.inject[Instruction, F](Forward(pos, l))
13        def backward(pos: Position, l: Int): Free[F, Position] = Free.inject[Instruction, F](Backward(pos, l))
14        def left(pos: Position, degree: Degree): Free[F, Position] = Free.inject[Instruction, F](RotateLeft(pos, degree))
15        def right(pos: Position, degree: Degree): Free[F, Position] = Free.inject[Instruction, F](RotateRight(pos, degree))
16        def showPosition(pos: Position): Free[F, Unit] = Free.inject[Instruction, F](ShowPosition(pos))
17      }
18
19      object Moves {
20        implicit def moves[F[_]](implicit I: Inject[Instruction, F]): Moves[F] = new Moves[F]
21      }
22
23      class PencilActions[F[_]](implicit I: Inject[PencilInstruction, F]) {
24        def pencilUp(pos: Position): Free[F, Unit] = Free.inject[PencilInstruction, F](PencilUp(pos))
25        def pencilDown(pos: Position): Free[F, Unit] = Free.inject[PencilInstruction, F](PencilDown(pos))
26      }
27
28      object PencilActions {
29        implicit def pencilActions[F[_]](implicit I: Inject[PencilInstruction, F]): PencilActions[F] = new PencilActions[F]
30      }
31    }
32  }
```

post.scala **view raw (https://gist.github.com/anonymous/0f788993b925f91805552ccc0acf435e/raw/c6491cc5d2c889cbb4486a9ce63b996300856928/post.scala) (https://gist.github.com/anonymous/0f788993b925f91805552ccc0acf435e#file-post-scala)** hosted with ❤ by **GitHub (https://github.com)**

`Moves` and `PencilActions` will be implicitly needed in our program. They gonna be parameterized by our `LogoApp` type, and will have all methods lifted to `Free` that will be operating on the `LogoApp`. That means we can mix them in one for comprehension expression. Now our program definition will look like this:

```
1   def program(implicit M: Moves[LogoApp], P: PencilActions[LogoApp]): (Position => Free[LogoApp, Unit]) = {
2       import M._, P._
3       s: Position =>
4       for {
5         p1 <- forward(s, 10)
6         p2 <- right(p1, Degree(90))
7         _ <- pencilUp(p2)
8         p3 <- forward(p2, 10)
9         _ <- pencilDown(p3)
10        p4 <- backward(p3, 20)
11        _ <- showPosition(p4)
12      } yield ()
13    }
```

post.scala **view raw (https://gist.github.com/anonymous/910f6a021798a9d0629552a1aacf6d05/raw/2a3a518e06f9d39b174412d2ec45cd0904e7dbf5/post.scala) (https://gist.github.com/anonymous/910f6a021798a9d0629552a1aacf6d05#file-post-scala)** hosted with ❤ by **GitHub (https://github.com)**

The last step is to add an interpreter for `PencilInstruction` and join it with previous one which is very easy thanks to functions offered by `cats`.

```
1   object PenInterpreterId extends (PencilInstruction ~> Id) {
2     def apply[A](fa: PencilInstruction[A]): Id[A] = fa match {
3       case PencilUp(p) => println(s"stop drawing at position $p")
4       case PencilDown(p) => println(s"start drawing at position $p")
5     }
6   }
```

post.scala **view raw (https://gist.github.com/anonymous/b074c4ee5d0b9068b619e8b0f185acd6/raw/865f8d6323d8bc28d782c52057b488032726c502/post.scala) (https://gist.github.com/anonymous/b074c4ee5d0b9068b619e8b0f185acd6#file-post-scala)** hosted with ❤ by **GitHub (https://github.com)**

This is our second interpreter and the code merging it with the existing one looks like this

```
1   val interpreter: LogoApp ~> Id = InterpreterId or PenInterpreterId
```

post.scala **view raw (https://gist.github.com/anonymous/a652e57ca7f140e48551bc5931618779/raw/12fbaaf9700919b675a710c18d025f4d7e75b4a0/post.scala) (https://gist.github.com/anonymous/a652e57ca7f140e48551bc5931618779#file-post-scala)** hosted with ❤ by **GitHub (https://github.com)**

As we can see, combining two sets of instruction is fairly easy. Mixing another one will be very similar, you just need to add another `Co product` which takes as type parameters the `LogoApp` and the other instruction set. Still, each of these DSLs can work separately and can be easily tested. That is a big advantage.

Summary

Summing up, Free Monad is definitely concept that make it possible to write functional code in easy and composable way. What is also very useful it helps you define your domain language and then just use it without bothering about implementation details. The code is readable and easily testable. The most difficult part is to grasp the theory and definitions that stays behind Free Monad. But it's worth learning.

Links

- Cats Free Monad tutorial (http://typelevel.org/cats/tut/freemonad.html)
- Eugene Yokota tutorial on Cats (http://eed3si9n.com/herding-cats/)
- Rúnar Bjarnason talk about free monad (https://www.youtube.com/watch?v=M258zVn4m2M)

You like this post? Want to stay updated? Follow us on Twitter (https://twitter.com/scalac_io) or subscribe to our Feed (/feeds/index.xml).

← (/2016/05/26/simple-types-in-play.html) → (/2016/06/23/each_lib_macros.html)