Free monads in practice

Wojciech Wiśniewski[1]

[1] FPComplete

March 2023

Problem

We need to register the user in the system.

To do this, we need to follow a sequence of steps:

- check whether there is a user account in service A
- set up an account in service B (or use an existing account)
- using the account identifier from service A, send a SMS with the PIN to the user
- receive PIN confirmation from the user
- ...

Problem continued

Each of these steps may fail (network errors, service restart, errors in the code...). In addition, our program can be restarted at any time.

We would like to be able to resume registration procedure from the point it was interrupted.

We would like our restartable program not to be too different from the ordinary one written in IO: we allow restartable operations to require annotations, but apart from that we want to use the usual do-notation.

Custom monad solution

Imagine we have type `Restartable a` and `Monad` instance for this type.

Lets imagine we have function `step :: String -> IO a -> Restartable a`, which we will use to label restartable operations.

Let's make a class and functions to write and read data from a file:

```
class Persistent a where
  serialize :: a -> String
  deserialize :: String -> a

save :: Persistent a => Handle -> a -> IO ()
save handle a = hPutStrLn handle $ serialize a

restore :: Persistent a => Handle -> IO (Maybe a)
restore handle = do
  eof <- hIsEOF handle
  if eof then
    pure Nothing
  else
    Just . deserialize <$> hGetLine handle
```

Custom monad solution continued

Imagine we also have function `runRestartable :: FilePath -> Restartable a -> IO a`, to run our probram in `IO`

Our own bespoke Restartable monad

Description of restartable operation:

```
data Restartable a where
  Step :: Persistent a => String -> IO a -> Restartable a

step :: String -> IO a -> Restartable a
step = Step
```

Our own bespoke Restartable monad: instances

We add constructors, which “capture” methods from `Monad` and `Applicative` classes:

```
data Restartable a where
  Step :: Persistent a => String -> IO a -> Restartable a
  Pure :: a -> Restartable a
  Bind :: Restartable x -> (x -> Restartable a) -> Restartable a

instance Applicative Restartable where
  pure = Pure

instance Monad Restartable where
  (>>=) = Bind

step :: String -> IO a -> Restartable a
step = Step
```

Other methods are defined in terms of `Monad` instance:

```
data Restartable a where
  Step :: Persistent a => String -> IO a -> Restartable a
  Pure :: a -> Restartable a
  Bind :: Restartable x -> (x -> Restartable a) -> Restartable a

instance Functor Restartable where
  fmap = liftM

instance Applicative Restartable where
  pure = Pure
  (<*>) = ap

instance Monad Restartable where
  (>>=) = Bind

step :: String -> IO a -> Restartable a
step = Step
```

Our own bespoke Restartable monad: interpreter

runRestartable function (notice that go function is recursive):

```
runRestartable :: forall a . FilePath -> Restartable a -> IO a
runRestartable path restartable = withFile path ReadWriteMode run
  where
    run :: Handle -> IO a
    run handle = go restartable
      where
        go :: Restartable b -> IO b
        go = \case
          Step name act -> do
            maybeA <- restore handle
            case maybeA of
              Just a -> do
                putStrLn $ "step " <> name <> " already completed"
                pure a
              Nothing -> do
                putStrLn $ "running step " <> name
                a <- act
                save handle a
                pure a
          Pure a ->
            pure a
          Bind act f ->
            go act >>= (go . f)
```

Simply “capturing” class methods can result in unlawfull instances.

Let’s see this in a simpler example of a free monoid:

```
class Monoid a where
  mempty :: a
  mappend :: a -> a -> a

data FreeMonoid a = Embed a | Mempty | Mappend (FreeMonoid a) (FreeMonoid a)

instance Monoid (FreeMonoid a) where
  mempty = Mempty
  mappend = Mappend
```

Digression about laws continued

`FreeMonoid` is violating the associativity law:

```
(a `mappend` b) `mappend` c /= a `mappend` (b `mappend` c)

        *            *
       / \          / \
      *   c   /=   a   *
     / \              / \
    a   b            b   c
```

It "remembers too much": we are only interested in the order of leaves in the tree, regardless of its construction history. Lawfull free monoid is a list:

```
instance Monoid [a] where
  mempty = []
  mappend = (++)
```

Our "Monad" instance for `Restartable` is also unlawfull, i. e. it violates this law:

```
pure a >>= f == f a
```

Free monad

We want to have lawfull monad.

In order to achieve this we separate the operations needed in every monad (pure, >>=) from those specific to our monad.

Common part:

```
data Free f a = Pure a
              | Impure (f (Free f a)) -- alternatively: Impure (Fix (Free f))

instance Functor f => Functor (Free f) where
  fmap = liftM -- implemented in terms of Monad

instance Functor f => Applicative (Free f) where
  pure = Pure
  (<*>) = ap -- implemented in terms Monad

instance Functor f => Monad (Free f) where
  (Pure a) >>= f = f a
  (Impure i) >>= f = Impure ((>>= f) <$> i)
```

Notice that `Monad` instance for `Free f` reqires `f` to be as `Functor`

Free monad continued

```haskell
liftF :: Functor f => f a -> Free f a
liftF command = Impure (Pure <$> command)

interpret :: (Functor f, Monad m) => (forall x. f x -> m x) -> Free f a -> m a
interpret nt = go
  where
    go = \case
      Pure a    -> pure a
      Impure fx -> nt fx >>= go
```

Free monad: operation specific to Restartable

Let's try to define `RestartableF` type:

```
type Restartable = Free RestartableF

data RestartableF a where
  Step :: Persistent a => String -> IO a -> Restartable a

instance Functor RestartableF where
  fmap f = ???
```

The `Monad` instance for `Free f` requires `f` to be a functor, but we can't turn `RestartableF` into a functor because of constraint `Persistent a`.

Is it a dead end?

Make functor from any type with this one weird trick

Let's use the same trick as at the beginning: let's "capture" the function provided in `fmap` in our data type:

```
data RestartableF next where
  Step :: Persistent a => String -> IO a -> (a -> next) -> Restartable next

instance Functor RestartableF where
  fmap g (Step name act f) = Step name act (g . f)

step :: Persistent a => String -> IO a -> Restartable a
step name act = liftF $ Step name act id
```

This trick is known under fancy name of co-Yoneda. This is the continuation passing style variant (note how `step` function sets an empty continuation `id`).

Free monad: interpreter

Note that `go` function is not recursive any more (we factored out recursion to `Free` type and `interpret` function):

```
runRestartable :: forall a . FilePath -> Restartable a -> IO a
runRestartable path restartable = withFile path ReadWriteMode run
  where
    run :: Handle -> IO a
    run handle = interpret go restartable
      where
        go :: RestartableF b -> IO b
        go = \case
          Step name act cont -> do
            maybeA <- restore handle
            case maybeA of
              Just a -> do
                putStrLn $ "step " <> name <> " already completed"
                pure $ cont a
              Nothing -> do
                putStrLn $ "running step " <> name
                a <- act
                save handle a
                pure $ cont a
```

Freer monad

We can build the co-Yoneda trick into our `Free(r)` type. Additional benefits:

- we get rid of the requirement that our `f` must be a functor
- we get rid of continuation passing style in our code

```
data Freer f a where
  Pure   :: a -> Freer f a
  Impure :: f x -> (x -> Freer f a) -> Freer f a

instance Monad (Freer f) where
  (Pure a)       >>= g = g a
  (Impure fx f) >>= g = Impure fx (f >=> g)

liftF :: f a -> Freer f a
liftF command = Impure command Pure

interpret :: Monad m => (forall x. f x -> m x) -> Freer f a -> m a
interpret nt = go
  where
    go = \case
      Pure a          -> pure a
      Impure fx cont -> nt fx >>= (go . cont)
```

Freer monad: operation specific to Restartable

`RestartableF` no longer need to be a functor and we get rid of the continuation passing style from our code:

```
data RestartableF a where
  Step :: Persistent a => String -> IO a -> RestartableF a

type Restartable = Freer RestartableF

step :: Persistent a => String -> IO a -> Restartable a
step name act = liftF $ Step name act
```

Freer monad: interpreter

```haskell
runRestartable :: forall a . FilePath -> Restartable a -> IO a
runRestartable path restartable = withFile path ReadWriteMode run
  where
    run :: Handle -> IO a
    run handle = interpret go restartable
      where
        go :: RestartableF b -> IO b
        go = \case
          Step name act -> do
            maybeA <- restore handle
            case maybeA of
              Just a -> do
                putStrLn $ "step " <> name <> " already completed"
                pure a
              Nothing -> do
                putStrLn $ "running step " <> name
                a <- act
                save handle a
                pure a
```

Free monad: analogy with list / tree

Free monad resembles list a little bit:

```
data List a = Nil | Cons a (List a)
```

Let's turn two arguments of `Cons` into single pair:

```
data List a = Nil | Cons (a, List a)
```

`(a,)` is as functor, let's factor it out:

```
data ListF f a = Nil | Cons (f (ListF f a))
type List a = ListF (a,) a
```

`ListF` is very similar to `Free`.

```
data ListF f a = Nil    | Cons   (f (ListF f a))
data Free  f a = Pure a | Impure (f (Free  f a))
```

You can think of `Free` as a tree. `Pure` values are the leaves, `Impure` are the nodes. Nodes have such a degree of branching as many "holes" there are in the `f` constructor contained within them. `ma >>= f` replaces all leaves in `ma` with the results of application of `f`.

Free monad: efficiency problems

Due to the "tree-like" structure of `Free`, the complexity of a single >>= operation is linear with respect to the size of the program (analogous to adding an element at the end of the list). Thus construction of the entire program has quadratic complexity.

Digression: Church encoding

Any algebraic data type can be encoded as a function:

- the result of the function is any (polymorphic) type `r`
- this function has as many arguments as there are constructors in the type
- each of these arguments is a function receiving the same arguments as its corresponding constructor
- the result of each of the arguments is the type `r`
- in particular, if the constructor has zero parameters, the argument is a single value of type `r`
- recursive arguments in constructors are replaced by `r`

Digression: Church encoding continued

```
data Bool = False | True
type ChurchBool = forall r. r -> r -> r
toChurch :: Bool -> ChurchBool
toChurch = \case
  False -> \onFalse onTrue -> onFalse
  True  -> \onFalse onTrue -> onTrue
fromChurch :: ChurchBool -> Bool
fromChurch f = f False True

data Either a b = Left a | Right b
type ChurchEither a b = forall r. (a -> r) -> (b -> r) -> r
toChurch :: Either a b -> ChurchEither a b
toChurch = \case
  Left a  -> \onLeft onRight -> onLeft a
  Right b -> \onLeft onRight -> onRight b
fromChurch :: ChurchEither a b -> Either a b
fromChurch f = f Left Right
```

```
data List a = Cons a (List a) | Nil
type ChurchList a = forall r. (a -> r -> r) -> r -> r
toChurch :: List a -> ChurchList a
toChurch = \case
  Cons a lst -> \onCons onNil -> onCons a ((toChurch lst) onCons onNil)
  Nil        -> \onCons onNil -> onNil
fromChurch :: ChurchList a -> List a
fromChurch f = f (:) []
```

Free monad: efficiency problems continued

Using Church encoding for `Free` elliminates quadratic complexity:

```
newtype ChurchFree f a =
  ChurchFree { runChurch :: forall r. (a -> r) -> (f r -> r) -> r }

instance Monad (ChurchFree f) where
  ChurchFree m >>= f = ChurchFree (\kp kf -> m (\a -> runChurch (f a) kp kf) kf
```