A Haskell free monad is a list of functors. Compare:

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

data Free f r = Pure r | Free (f (Free f r))

Pure is analogous to Nil and Free is analogous to Cons. A free monad stores a list of functors instead of a list of values. Technically, you could implement free monads using a different data type, but any implementation should be isomorphic to the above one.

You use free monads whenever you need an abstract syntax tree. The base functor of the free monad is the shape of each step of the syntax tree.

[My post](http://www.haskellforall.com/2012/06/you-could-have-invented-free-monads.html), which somebody already linked, gives several examples of how to build abstract syntax trees with free monads

**Why free monads matter**

**Interpreters**

Good programmers decompose data from the interpreter that processes that data. Compilers exemplify this approach, where they will typically represent the source code as an abstract syntax tree, and then pass that tree to one of many possible interpreters. We benefit from decoupling the interpreter and the syntax tree, because then we can interpret the syntax tree in multiple ways. For example, we could:

* compile it to an executable,
* run it directly (i.e. the traditional sense of "interpret"),
* pretty print it,
* compress and archive it,
* or do nothing at all with it!

Each of those options corresponds to a different interpreter.  
  
Let's try to come up with some sort of abstraction that represents the essence of a syntax tree. Abstractions always begin from specific examples, so let's invent our own toy programming language and try to represent it as a data type.  
  
Our toy language will only have three commands:

output b -- prints a "b" to the console

bell -- rings the computer's bell

done -- end of execution

So we represent it as a syntax tree where subsequent commands are leaves of prior commands:

data Toy b next =

Output b next

| Bell next

| Done

Notice how the Done command has no leaf since it must be the last command.  
  
Then I could write a sample program that I might want to pass to an interpreter:

-- output 'A'

-- done

Output 'A' Done :: Toy Char (Toy a next)

... but unfortunately this doesn't work because every time I want to add a command, it changes the type:

-- bell

-- output 'A'

-- done

Bell (Output 'A' Done) :: Toy a (Toy Char (Toy b next)))

Fortunately, we can cheat and use the following data type to wrap as many Toys as we want into the same data type:

data Cheat f = Cheat (f (Cheat f))

With Cheat we've defined a stream of functors that will only end when it gets to the Done constructor. Fortunately, Cheat already exists in Haskell and goes by another name:

data Fix f = Fix (f (Fix f))

It's named Fix because it is "the fixed point of a functor".  
  
With Fix in hand, now we can fix our example programs:

Fix (Output 'A' (Fix Done)) :: Fix (Toy Char)

Fix (Bell (Fix (Output 'A' (Fix Done)))) :: Fix (Toy Char)

Now they have the same type. Perfect! Or is it?  
  
There's still a problem. This approach only works if you can use the Done constructor to terminate every chain of functors. Unfortunately, programmers don't often have the luxury of writing the entire program from start to finish. We often just want to write subroutines that can be called from within other programs and our Fix trick doesn't let us write a subroutine without terminating the entire program.  
  
Ok, so let's hack together a quick and dirty fix to work around this problem. Our subroutine finished but we are not ready to call Done, so instead we throw an exception and let whoever calls our subroutine catch it and resume from where we left off:

data FixE f e = Fix (f (FixE f e)) | Throw e

Then we write a catch function:

catch ::

(Functor f) => FixE f e1 -> (e1 -> FixE f e2) -> FixE f e2

catch (Fix x) f = Fix (fmap (flip catch f) x)

catch (Throw e) f = f e

We can only use this if Toy b is a functor, so we muddle around until we find something that type-checks (and satisfies the [Functor laws](http://hackage.haskell.org/packages/archive/base/4.3.1.0/doc/html/Control-Monad.html#t:Functor)):

instance Functor (Toy b) where

fmap f (Output x next) = Output x (f next)

fmap f (Bell next) = Bell (f next)

fmap f Done = Done

Now we can write code that can be caught and resumed:

data IncompleteException = IncompleteException

-- output 'A'

-- throw IncompleteException

subroutine = Fix (Output 'A' (Throw IncompleteException))

:: FixE (Toy Char) IncompleteException

-- try {subroutine}

-- catch (IncompleteException) {

-- bell

-- done

-- }

program = subroutine `catch` (\\_ -> Fix (Bell (Fix Done))

:: FixE (Toy Char) e

**Free Monads - Part 1**

So we proudly package up this "improved" Fix and release it on Hackage under the package name fix-improved, and then find out that the users are misusing the library. They start using the exception to pass around ordinary values instead of exceptional values. How dare they! Exceptions are only for **exceptional** situations and not for ordinary flow control. What a bunch of morons!  
  
... except we are the morons, because our FixE already exists, too, and it's called the [Free monad](http://hackage.haskell.org/packages/archive/free/2.0.3/doc/html/Control-Monad-Free.html#t:Free):

data Free f r = Free (f (Free f r)) | Pure r

As the name suggests, it is automatically a monad (if f is a functor):

instance (Functor f) => Monad (Free f) where

return = Pure

(Free x) >>= f = Free (fmap (>>= f) x)

(Pure r) >>= f = f r

The return was our Throw, and (>>=) was our catch. Our users were actually using the e values as return values because that is the correct way to use them within a monad.  
  
The great part about Haskell is that for any monad we get do notation for free. However, Free (Toy b) is the monad, not Toy b, which means that if we want to sequence our primitive commands using do notation, we have convert our commands of type Toy b into Free (Toy b). Our attempt to do so produces something that looks like this:

output :: a -> Free (Toy a) ()

output x = Free (Output x (Pure ()))

bell :: Free (Toy a) ()

bell = Free (Bell (Pure ()))

done :: Free (Toy a) r

done = Free Done

I'll be damned if that's not a common pattern we can abstract:

liftF :: (Functor f) => f r -> Free f r

liftF command = Free (fmap Pure command)

output x = liftF (Output x ())

bell = liftF (Bell ())

done = liftF Done

Now, we can sequence these primitive commands using do notation, and everything just works! Let's translate our previous example, getting rid of the superfluous exceptions:

subroutine :: Free (Toy Char) ()

subroutine = output 'A'

program :: Free (Toy Char) r

program = do

subroutine

bell

done

This is where things get magical. We now have do notation for something that hasn't even been interpreted yet: it's pure data. Newcomers to Haskell often associate monads with side effects or actions, but the above code does nothing more than build a data type. We can prove that it is still just an ordinary data type by defining a function to convert it to a string:

showProgram :: (Show a, Show r) => Free (Toy a) r -> String

showProgram (Free (Output a x)) =

"output " ++ show a ++ "\n" ++ showProgram x

showProgram (Free (Bell x)) =

"bell\n" ++ showProgram x

showProgram (Free Done) =

"done\n"

showProgram (Pure r) =

"return " ++ show r ++ "\n"

.. and printing it:

>>> putStr (showProgram program)

output 'A'

bell

done

It looks like we just inadvertently defined our first interpreter: the pretty printer! We can use our pretty printer to quickly check that our monad obeys some of the [monad laws](http://hackage.haskell.org/packages/archive/base/4.3.1.0/doc/html/Control-Monad.html#t:Monad):

pretty :: (Show a, Show r) => Free (Toy a) r -> IO ()

pretty = putStr . showProgram

>>> pretty (output 'A')

output 'A'

return ()

>>> pretty (return 'A' >>= output)

output 'A'

return ()

>>> pretty (output 'A' >>= return)

output 'A'

return ()

>>> pretty ((output 'A' >> done) >> output 'C')

output 'A'

done

>>> pretty (output 'A' >> (done >> output 'C'))

output 'A'

done

Notice how Done swallows all commands after it, unlike Pure. I only included Done in the Toy functor for illustrative purposes. In many cases you don't need a Done-like constructor in your functor since you probably want Pure's resumable behavior, however in other cases you may actually want Done's "abort" semantics.  
  
We could also write an actual interpreter in the conventional sense of the word:

ringBell :: IO () -- some obnoxious library would provide this

interpret :: (Show b) => Free (Toy b) r -> IO ()

interpret (Free (Output b x)) = print b >> interpret x

interpret (Free (Bell x)) = ringBell >> interpret x

interpret (Free Done ) = return ()

interpret (Pure r) = throwIO (userError "Improper termination")

The free monad is completely agnostic as to how it is used.

**Concurrency**

Let's say we have two monadic "threads" we want to interleave. For IO, we could just use forkIO to run them in parallel, but what if we wanted to thread two State monads or even two Cont monads. How would that even work?  
  
Well, we could try representing a thread as a list of individual monad actions.

type Thread m = [m ()]

... but this doesn't guarantee that our interpreter will call them in the order we list them, nor does it allow us to pass return values between successive monad actions. We can enforce their ordering, though, by nesting each subsequent action within the previous one, and if there are no more actions left, we use a separate constructor to indicate we are done:

data Thread m r = Atomic (m (Thread m r)) | Return r

This nesting forces the first action to be evaluated before the next one can be revealed and the Atomic constructor wraps one indivisible step. We can then turn any single monad invocation into an atomic Thread step:

atomic :: (Monad m) => m a -> Thread m a

atomic m = Atomic $ liftM Return m

Now we need a way to make Thread a monad, but we will just "pretend" that we sequence two threads while still keeping their atomic steps separate so that we can later interleave them with other threads.

instance (Monad m) => Monad (Thread m) where

return = Return

(Atomic m) >>= f = Atomic (liftM (>>= f) m)

(Return r) >>= f = f r

Using this, we can write threads broken into atomic steps:

thread1 :: Thread IO ()

thread1 = do

atomic $ print 1

atomic $ print 2

thread2 :: Thread IO ()

thread2 = do

str <- atomic $ getLine

atomic $ putStrLn str

All we are missing is a way to interleave two threads, while still maintaining the atomicity of the individual steps. Let's just do a naive alternation:

interleave ::

(Monad m) => Thread m r -> Thread m r -> Thread m r

interleave (Atomic m1) (Atomic m2) = do

next1 <- atomic m1

next2 <- atomic m2

interleave next1 next2

interleave t1 (Return \_) = t1

interleave (Return \_) t2 = t2

Now we need a way to run threads after we are done interleaving them:

runThread :: (Monad m) => Thread m r -> m r

runThread (Atomic m) = m >>= runThread

runThread (Return r) = return r

>>> runThread (interleave thread1 thread2)

1

[[Input: "Hello, world!"]]

2

Hello, world!

Magic! We just wrote a primitive threading system in Haskell! Now try using it with the pure State monad.

**Free Monads - Part 2**

If you've been paying attention, Thread is just Free in disguise and atomic is liftF. The above example shows how a free monad greatly resembles a list. In fact, just compare the definition of Free to the definition of a List:

data Free f r = Free (f (Free f r)) | Pure r

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

In other words, we can think of a free monad as just being a list of functors. The Free constructor behaves like a Cons, prepending a functor to the list, and the Pure constructor behaves like Nil, representing an empty list (i.e. no functors).  
  
So if a List is a list of values, and a free monad is just a list of functors, what happens if the free monad's functor is itself a value:

type List' a = Free ((,) a) ()

List' a

= Free ((,) a) ()

= Free (a, List' a)) | Pure ()

= Free a (List' a) | Pure ()

It becomes an ordinary list!  
  
A list is just a special case of a free monad. However, the Monad instance for [] is not the same thing as the Monad instance for List' a (i.e. Free ((,) a)). In the List' a monad, join behaves like (++) and return behaves like [], so you can think of the List' a monad as just being a fancy way to concatenate values using do notation.  
  
When you think of free monads as lists, a lot of things become much more obvious. For example, liftF is just like the singleton list, creating a free monad with exactly one functor in it:

singleton x = Cons x Nil -- i.e. x:[], or [x]

liftF x = Free (fmap Pure x)

Similarly, our interleave function is just a list merge:

merge (x1:xs1) (x2:xs2) = x1:x2:merge xs1 xs2

merge xs1 [] = xs1

merge [] xs2 = xs2

-- this is actually more similar to:

-- [x1] ++ [x2] ++ interleave xs1 xs2

interleave (Atomic m1) (Atomic m2) = do

next1 <- liftF m1

next2 <- liftF m2

interleave next1 next2

interleave a1 (Return \_) = a1

interleave (Return \_) a2 = a2

So really, when you think of it that way, concurrency is nothing more than merging a bunch of lists of actions. In a later post, I will review a great paper that demonstrates how you can actually build elegant and robust threading systems and schedulers using this free monad approach.  
  
It's not a coincidence that free monads resemble lists. If you learn category theory, you'll discover that they are both free objects, where lists are free monoids, and free monads are ... well, free monads.

**Interpreters - Revisited**

In the first section I presented the concept of using free monads for interpreters, but the concept of an interpreter is more powerful and useful than it sounds and it's not just limited to compilers and pretty printers.  
  
For example, let's say you wanted to one-up Notch's game idea for 0x10c and make a player-programmable game ... except in Haskell! You want to accept programs from players that they can run in the game, but you don't want to give them full-blown access to the IO monad, so what do you do?  
  
The naive approach might be to copy the Haskell language's original design, where output is presented as list of requests made to the outside world and input is presented as a list of responses received from the outside world:

main :: [Response] -> [Request]

The Request type would enumerate the sort of actions you could take and the Response type would delimit the results you would get back. Then for our game, the set of inputs might be:

data Request =

Look Direction

| ReadLine

| Fire Direction

| WriteLine String

... and the responses might be:

data Response =

Image Picture -- Response for Look

| ChatLine String -- Response for Read

| Succeeded Bool -- Response for Write

Well, that certainly won't work. There is no clear coupling between requests and responses (Fire doesn't even have a response), and it's not clear what should happen if you try to read responses before you even generate requests.  
  
So let's try to impose some kind of order on these inputs and outputs by merging them into a single data type:

data Interaction next =

Look Direction (Image -> next)

| Fire Direction next

| ReadLine (String -> next)

| WriteLine String (Bool -> next)

Each constructor can have some fields the player fills in (i.e. the player's requests), and they can also provide functions which the interpreter will supply input to. You can think of this Interaction type as the contract between the programmer and the interpreter for a single step.  
  
Conveniently, Interaction forms a functor:

instance Functor Interaction where

fmap f (Look dir g) = Look dir (f . g)

fmap f (Fire dir x) = Fire dir (f x)

fmap f (ReadLine g) = ReadLine (f . g)

fmap f (WriteLine s g) = WriteLine s (f . g)

Actually, you don't even have to write that. GHC provides the DeriveFunctor extension, which would you let you just write:

data Interaction ... deriving (Functor)

... and it will get it correct.  
  
As always, we can create a list of actions by using the Free monad:

type Program = Free Interaction

With Program in hand, the player can now write a simple program:

easyToAnger = Free $ ReadLine $ \s -> case s of

"No" -> Free $ Fire Forward

$ Free $ WriteLine "Take that!" (\\_ -> easyToAnger)

\_ -> easyToAnger

The interpreter can then interpret the program for him, perhaps converting it into some sort of Game monad:

interpret :: Program r -> Game r

interpret prog = case prog of

Free (Look dir g) -> do

img <- collectImage dir

interpret (g img)

Free (Fire dir next) -> do

sendBullet dir

interpret next

Free (ReadLine g) -> do

str <- getChatLine

interpret (g str)

Free (WriteLine s g) ->

putChatLine s

interpret (g True)

Pure r -> return r

Every free monad is guaranteed to be a monad, so we can always give the player syntactic sugar for writing their programs using Haskell do notation:

look :: Direction -> Program Image

look dir = liftF (Look dir id)

fire :: Direction -> Program ()

fire dir = liftF (Fire dir ())

readLine :: Program String

readLine = liftF (ReadLine id)

writeLine :: String -> Program Bool

writeLine s = liftF (WriteLine s id)

Now, the player can more easily write their program as:

easyToAnger :: Program a

easyToAnger = forever $ do

str <- readLine

when (str == "No") $ do

fire Forward

-- Ignore the Bool returned by writeLine

\_ <- writeLine "Take that!"

return ()

In short, we've given the player a sand-boxed interaction language that delimits their actions, yet complete with all the syntactic monad sugar and luxuries of programming in Haskell. On top of this, we've given ourselves the complete freedom to interpret the player's program any way we please. For example, if I were to release a patch tomorrow that changed the game world (and Haskell had some form of code hot-swapping), I could keep running the players' programs without interruption by just switching out the interpreter. Or, if I were sadistic, I could use the most aggressive player's program to control a real-world destructive robot of doom (a.k.a. the IO monad) and watch it wreak havoc.

**Free Monads - Part 3**

The free monad is the interpreter's best friend. Free monads "free the interpreter" as much as possible while still maintaining the bare minimum necessary to form a monad.  
  
Free monads arise every time an interpreter wants to give the program writer a monad, and nothing more. If you are the interpreter and I am the program writer, you can push against me and keep your options as free as possible by insisting that I write a program using a free monad that you provide me. The free monad is guaranteed to be the formulation that gives you the most flexibility how to interpret it, since it is purely syntactic.  
  
This notion of "freeing the interpreter" up as much as possible sounds a lot like an optimization problem, which you might phrase as follows:

What is the most flexible monad to interpret, given the constraint that it still must be a monad?

In fact, maximizing some notion of "freeness" given a constraint is the intuition that leads to the category theory definition of a [free object](http://en.wikipedia.org/wiki/Free_object), where the concept of "freeness" is made rigorous. A free monad just happens to be the "free-est" object that still forms a monad.

class Functor f whereSource

The Functor class is used for types that can be mapped over. Instances of Functor should satisfy the following laws:

fmap id == id

fmap (f . g) == fmap f . fmap g

The instances of Functor for lists, Data.Maybe.Maybe and System.IO.IO satisfy these laws.

Methods

fmap :: (a -> b) -> f a -> f bSource

Instances

Functor []

Functor IO

Functor [::]

Functor Maybe

Functor ReadP

Functor ReadPrec

Functor STM

Functor ZipList

Functor Id

Functor ((->) r)

Functor (Either a)

Functor ((,) a)

Functor (ST s)

Ix i => Functor (Array i)

Monad m => Functor (WrappedMonad m)

Functor (Const m)

Functor (StateR s)

Functor (StateL s)

Functor (ST s)

Arrow a => Functor (WrappedArrow a b)

data Free f a Source

Constructors

Pure a

Free (f (Free f a))

Instances

MonadTrans Free

(Functor m, MonadError e m) => MonadError e (Free m)

(Functor m, MonadReader e m) => MonadReader e (Free m)

(Functor m, MonadState s m) => MonadState s (Free m)

(Functor m, MonadWriter e m) => MonadWriter e (Free m)

Functor f => MonadFree f (Free f)

Functor f => Monad (Free f)

Functor f => Functor (Free f)

(Functor v, MonadPlus v) => MonadPlus (Free v)

Functor f => Applicative (Free f)

Foldable f => Foldable (Free f)

Traversable f => Traversable (Free f)

Alternative v => Alternative (Free v)

(Functor m, MonadCont m) => MonadCont (Free m)

Traversable1 f => Traversable1 (Free f)

Foldable1 f => Foldable1 (Free f)

Functor f => Apply (Free f)

Functor f => Bind (Free f)

(Eq (f (Free f a)), Eq a) => Eq (Free f a)

(Ord (f (Free f a)), Ord a) => Ord (Free f a)

(Read (f (Free f a)), Read a) => Read (Free f a)

(Show (f (Free f a)), Show a) => Show (Free f a)

class Monad m where

The Monad class defines the basic operations over a monad, a concept from a branch of mathematics known as category theory. From the perspective of a Haskell programmer, however, it is best to think of a monad as an abstract datatype of actions. Haskell's do expressions provide a convenient syntax for writing monadic expressions.

Minimal complete definition: >>= and return.

Instances of Monad should satisfy the following laws:

return a >>= k == k a

m >>= return == m

m >>= (\x -> k x >>= h) == (m >>= k) >>= h

Instances of both Monad and Functor should additionally satisfy the law:

fmap f xs == xs >>= return . f

The instances of Monad for lists, Data.Maybe.Maybe and System.IO.IO defined in the Prelude satisfy these laws.

Methods

(>>=) :: forall a b. m a -> (a -> m b) -> m b

Sequentially compose two actions, passing any value produced by the first as an argument to the second.

(>>) :: forall a b. m a -> m b -> m b

Sequentially compose two actions, discarding any value produced by the first, like sequencing operators (such as the semicolon) in imperative languages.

return :: a -> m a

Inject a value into the monadic type.

fail :: String -> m a

Fail with a message. This operation is not part of the mathematical definition of a monad, but is invoked on pattern-match failure in a do expression.