

Haskell for all

Saturday, June 9, 2012

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](#)):

```
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](#):

```
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](#):

```
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 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](#), where the concept of "freeness" is made rigorous. A free monad just happens to be the "free-est" object that still forms a monad.

Posted by [Gabriella Gonzalez](#) at [2:30 PM](#)



43 comments:

Anonymous [August 3, 2012 at 3:32 PM](#)

Great Article! The best I've found regarding an explanation of how the Free monad works

[Reply](#)



Cristian Petrescu-Prahova August 24, 2012 at 3:32 PM

This is awesome. How does the machinery work for non-linear languages, for example data Expr a = Value a | Plus a a?

[Reply](#)

▼ [Replies](#)



Gabriella Gonzalez August 24, 2012 at 4:21 PM

It forks the current context. For example, let's say I write:

```
forkPlus :: Free Expr Bool
forkPlus = liftF $ Plus False True
```

Then it would behave just like C's fork implementation, where the return value tells you which branch of the computation you are on:

```
do
  bool <- forkPlus
  if bool
  then ... -- On the right branch
  else ... -- On the left branch
```

Of course, you don't have to do it that way. You can always still use the conventional way to build the AST without the monad instance, by just defining:

```
plus :: Free Expr a -> Free Expr a -> Free Expr a
plus e1 e2 = Free $ Plus e1 e2
```

[Reply](#)



jberryman August 24, 2012 at 4:21 PM

Glad I finally bit the bullet and read my tabs on Free monads. I think they're what I've been looking for for a number of applications. Great writing!

[Reply](#)

Jesse September 13, 2012 at 2:48 AM



Excellent topic. Excellent writing. Every time I come back to haskell I generally spend a few days hammering boilerplate monad instantiation back into my head. I was also unaware that deriving functors was possible. You saved me a few days and then some :).

[Reply](#)



Ittay Dror [September 17, 2012 at 12:05 PM](#)

In `(Free x) >>= f = Free (fmap (>>= f) x)`, shouldn't there be a 'flip' before `>>=` ?

[Reply](#)

▼ [Replies](#)



Gabriella Gonzalez  [September 17, 2012 at 12:54 PM](#)

No, it is correct. I think it is more clear if I expand out the definition a little bit:

$$(Free\ x)\ >>= f = Free\ (fmap\ (\lambda m \rightarrow m\ >>= f)\ x)$$

Haskell's section syntax says that if `(*)` is some operator, then:

$$(*\ b) = \lambda a \rightarrow a\ *\ b$$

$$(a\ *) = \lambda b \rightarrow a\ *\ b$$

This can be a bit confusing because if you were to instead put the parentheses around the operator, then the operator is interpreted using prefix notation instead of section notation:

$$(*)\ a\ b = a\ *\ b$$

... which is confusing since `(* a)` and `(*) a` look very similar, but have different meanings:

$$(*\ a) = \lambda x \rightarrow x\ *\ a$$

$$(*)\ a = \lambda x \rightarrow a\ *\ x$$

[Reply](#)



basman October 18, 2012 at 4:34 PM

The instance definition of fmap for Toy b seems to reduce to saying that fmap f is just the identity function for any f. Is that right?

[Reply](#)

▼ [Replies](#)



Gabriella Gonzalez October 18, 2012 at 5:32 PM

Actually, it does change the Toy b x. Note that the functor definition is not recursive.

[Reply](#)



Mikhail Glushenkov March 18, 2013 at 9:08 AM

Great article, thanks!

[Reply](#)

Anonymous May 22, 2013 at 1:03 AM

Your initial example is distracting to me. You say you can't add commands to the program because the type always changes but actually the only reason the type changes is that you specify "nextstep" as a type parameter when it will actually only ever be Toy a. So the way you could "cheat" would be to define Toy in the normal way:

Toy a = Output a (Toy a) | Bell (Toy a) | Done

[Reply](#)

▼ [Replies](#)



Gabriella Gonzalez May 22, 2013 at 12:19 PM

The target audience of my post is people who have never been introduced to the idea of using a recursive data type, particularly a functor, to model commands. These people typically come from an imperative background that distinguishes statements and expressions. This is why the introduction explicitly dwells on that particular wrong way of doing things.



Unknown April 11, 2014 at 3:45 AM

This comment has been removed by the author.

[Reply](#)

Anonymous July 11, 2013 at 9:22 AM

The section on Interaction deserves its own post. What do we do when we want to implement a UI for the game instead of an AI? I imagine we'd do something like the following:

Define a type of Plans over some other computation, like Program above

```
type Plan = FreeT Interaction
```

Provide syntactic sugar for building a Plan:

```
liftMethod :: (Functor f, Monad m) => ((a -> FreeT f m a) -> f (FreeT f m a)) ->
FreeT f m a
```

```
liftMethod f = FreeT (return (Free (f return)))
```

```
look :: (Monad m) => Direction -> Plan m (Image)
```

```
look direction = liftMethod (Look direction)
```

```
liftFT :: (Functor f, Monad m) => (FreeT f m () -> f (FreeT f m ())) -> FreeT f m ()
```

```
liftFT a = FreeT (return (Free (a (return ())))))
```

```
fire :: (Monad m) => Direction -> Plan m ()
```

```
fire direction = liftFT (Fire direction)
```

```
readLine :: (Monad m) => Plan m (String)
```

```
readLine = liftFT (ReadLine)
```

```
writeLine :: (Monad m) => String -> Plan m (Bool)
```

```
writeLine message = liftMethod (WriteLine message)
```

I think liftFT could be rewritten in terms of liftF, or fire could be defined as:

```
fire direction = liftF (Fire direction id)
```

I haven't figured out a more elegant liftMethod.

Then our UI could be written as

```
ui :: Direction -> Plan IO ()
```

```
ui initialDirection = do ...
```

Code to look, display an image, check for a chat message, display a chat, get input, and decide to change what direction we are looking / fire / or send a chat

I'm not sure how we can poll for a new incoming chat message with ReadLine, since it seems like it would block until there is a String to pass to next. Perhaps it should be:

```
data Interaction next =
```

```
Look Direction (Image -> next)
```

```
| Fire Direction next
```

```
| ReadLine (Maybe String -> next)
```

```
| WriteLine String next
```

[Reply](#)

▼ [Replies](#)



Gabriella Gonzalez  July 11, 2013 at 9:29 AM

You are echoing the same line of thinking that led me to formulate my `pipes` library. I was trying to figure out how to use the same abstraction for both building the DSL and consuming the DSL. The result was to use a free monad to produce events (i.e. a Producer) and a free monad to consume events (i.e. a Consumer) and both were special cases of a Pipe.

However, there is some loss of power as a result, so take this answer with a grain of salt.

[Reply](#)



memetic warrior January 8, 2014 at 6:07 AM

Fantastic article. I appreciate your "historic" approach mimicking a incremental line of reasoning . That is how people learn and the knowledge can be transmitted, by example and incrementally, not by enumeration of mathematical results. The latter is

good to store knowledge, not to transmit it. The process of discovery must be reconstructed in each one's mind to fully grasp the meaning. Why force people to rediscover the path if the author know it and can transmit it? Maybe one day transmission of knowledge can be formalized.

Less philosophical: I suppose that free monad interpreters is what Oleg and others use to modelize effects isn't it?. Do you plan to explain effects that way if you didn't already?

[Reply](#)

▼ [Replies](#)



Gabriella Gonzalez  January 11, 2014 at 8:57 PM

Yes, I agree that seeing the fully formed solution is not as helpful as seeing the thought-process that led to it. It also helps motivate why the solution exists in the first place.

Free monad interpreters are definitely one approach to modeling effects, but this is not a subject I'm an expert on. The last time I checked, one of Conor McBride's students, Stevan Andjelkovic was working seriously on taking this to its logical conclusion but I haven't followed up on that. There are lots of subtle details.

[Reply](#)



James Moore March 28, 2014 at 12:38 PM

It's named Fix because it is "the fixed point of a functor".

I'd never heard the term "fixed point of a functor" before this article, FYI. I'm off googling it now, but you might want to explain that here.

(Update after a few minutes with google: the articles you get when you google "fixed point of a functor" are pretty beginner-hostile.)

[Reply](#)

▼ [Replies](#)



Gabriella Gonzalez  April 9, 2014 at 5:37 PM

The best resource I have read for fixed points of functors is this one:

<http://homepages.inf.ed.ac.uk/wadler/papers/free-rectypes/free-rectypes.txt>

However, even that is a bit dense. I am slowly working on writing up an explanation of my own.

[Reply](#)



Unknown February 5, 2015 at 3:00 AM

Hi Gabriel,

Regarding:

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

Here is a working implementation (at least on OS X) based on a not so obnoxious library ;)

```
ringBell = putStr "\a"
```

Thanks for this enlightening article!

Daniel

[Reply](#)



Unknown February 20, 2015 at 2:38 PM

Hi Gabriel,

I cannot figure out what you mean with sort of "Game" monad. The only thing I was able to do was:

```
type Game = IO
```

but I would like to have a way to define the functions "collectImage", "sendBullet", etc ... in a type class and then provide the instance with the implementation.

Could you make a short example?

Thanks for the beautiful article!

Matteo

[Reply](#)

▼ [Replies](#)



Gabriella Gonzalez  [March 1, 2015 at 12:47 PM](#)

So usually ``Game`` is almost always ``IO``. I was just playing it safe by leaving it unspecified.

However, note that you don't gain much by translating the free monad to some type classed monad (i.e. some ``MonadGame`` type class). The reason why is that the type class encodes the same amount of semantic information as the original free monad: none at all. Actually, the type class is a little less powerful because now you no longer have access to the syntax tree any longer once you translate it to the type class and you can't perform manipulations on the syntax tree.

Most of the time you don't really need to manipulate the syntax tree anyway, so it's not a huge loss, but there are some cases where it does come in handy. An example of a library that heavily takes advantage of syntax tree manipulation is my ``pipes`` library, where all ``pipes`` are basically syntax trees of ``Request`/`Respond`` commands, and when you compose two pipes together it is fusing the two syntax trees together into a new syntax tree.

[Reply](#)



Vlad Patryshev [October 3, 2015 at 7:30 PM](#)

This comment has been removed by the author.

[Reply](#)



Vlad Patryshev [October 3, 2015 at 7:49 PM](#)

Gabriel, thanks for the great text; but in the beginning of it you call the argument of `Fix` (and of `Cheat`) a functor. It is not obvious at all why it is a functor. Later you explain that you need `fmap`; that's ok, but that's confusing.

[Reply](#)



Unknown [October 8, 2015 at 9:36 PM](#)

Great post! Thank you!

[Reply](#)



wakalabis March 17, 2016 at 5:37 PM

Beautiful post. Thanks! Is it possible to create a BASIC-like language using Free Monads? I can imagine writing one, but I don't know an easy way to implement GOTO and GOSUB commands.

[Reply](#)

▼ [Replies](#)



Gabriella Gonzalez March 20, 2016 at 12:01 PM

Actually, it looks like somebody figured out how to implement `GOTO` using a free monad here:

<http://d.hatena.ne.jp/fumiexcel/20121111/1352624678>



wakalabis March 21, 2016 at 2:14 PM

Nice find! Thanks! Now am trying to figure out a clean way to make forward jumps.



wakalabis March 21, 2016 at 2:37 PM

Never mind. There's an example on the same blog showing how to jump forward!

<http://d.hatena.ne.jp/fumiexcel/20121111/1352642335>

[Reply](#)



Mark July 23, 2016 at 3:46 PM

So, I'm a bit of a Haskell newbie and I don't understand your Cheat - what it does or how it solves the problem. I looked up Fix (actually found a description of 'fix' - lowercase) but don't really see the point there either.

[Reply](#)



E.R.I.K. August 5, 2016 at 2:19 PM

This comment has been removed by the author.

[Reply](#)



E.R.I.K. August 5, 2016 at 2:25 PM

Regarding the figuring out of GOTO: funny I dropped you a note at your "Data is Code" artikel mentioning libraries.. the same guy wrote those libraries and the GOTO.. genius.. i recognise the use of "piyo" and "hoge" as names.. must be.. <http://www.haskellforall.com/2016/04/data-is-code.html?showComment=1467898064772#c3710112311512301556>

[Reply](#)



adam November 26, 2016 at 7:58 PM

Thinking of Free Monad as a Linked List with the each node value as the Functor, and the tail being the Free instance contained within the functor...

It finally clicked for me!

[Reply](#)



M@rk1x April 1, 2017 at 12:41 AM

Hi Gabriel,

thanks for writing this article. Though I am still trying to figure it out, it looks well-written.

One thing I am struggling with, is why you used the type "forall b next, Toy b next" which then needs to be "Fix"ed. Couldn't you use forall b, Toy b and be done with it?

[Reply](#)

▼ [Replies](#)



M@rk1x April 1, 2017 at 1:21 AM

Oh I see, you want to have different types of outputs, but still you could use GADTs as follows:

```
data ToyG a where
  OutG :: b -> ToyG b
  BellG :: ToyG b -> ToyG b
  DoneG :: ToyG a
```

then again, I suppose, yours is a fine way to define GADTs out of normal recursive types.



M@rk1x April 1, 2017 at 12:19 PM

wrong again. To answer also a question above. The whole point of 'next' is to provide a recursive variable to the recursive operator Free. If the Toy had been defined as I said, you probably could have used Toy as a Free Monad.



Gabriella Gonzalez April 4, 2017 at 1:38 PM

Yeah, the `next` parameter is basically where the recursion goes. For example, suppose that you wrote the `Toy` type in the following more direct form without using `Fix`:

```
data Toy b
  __ = Output b (Toy b)
  __| Bell (Toy b)
  __| Done
```

Then you just replace every recursive occurrence of the `Toy` data type with `next` to get the data type that you would pass to `Fix`:

```
data Toy b next
  __ = Output b next
  __| Bell next
  __| Done
```

Reply



Unknown April 24, 2017 at 8:00 PM

yield = atomic \$ return ()

[Reply](#)



boris November 12, 2017 at 7:39 AM

Gabriel thanks a lot for this and all your posts. Part1 states "We now have do notation for something that hasn't even been interpreted yet: it's pure data". It means that data retrieved from some store and network can be also interpreted. But it (pure data) seems to be no longer true for example in Part3. My question is what are conditions for Free to stay "pure data". Intuitively seems that as long as underlying functors are pure data (as opposite to functions) Free is also pure data. Is it correct? Thanks

[Reply](#)

▼ Replies



Gabriella Gonzalez November 12, 2017 at 8:46 AM

Yes, that is correct



boris November 14, 2017 at 8:20 PM

Thanks

Reply



Janus Troelsen June 7, 2021 at 9:21 PM

"In a later post, I will review a great paper that demonstrates how you can actually build elegant and robust threading systems"

Which post is that? Has it been written yet? I'd appreciate a link.

[Reply](#)

▼ Replies



Gabriella Gonzalez June 7, 2021 at 9:52 PM

The closest post is this one: <https://www.haskellforall.com/2013/06/from-zero-to-cooperative-threads-in-33.html>

I ended up not structuring the post as a paper review, but it's basically based on that paper that I had in mind when I said that.

[Reply](#)

To leave a comment, click the button below to sign in with Google.

SIGN IN WITH GOOGLE



[Newer Post](#) • • • • • [Home](#) • • • • • [Older Post](#)

Subscribe to: [Post Comments \(Atom\)](#)

Blog Archive

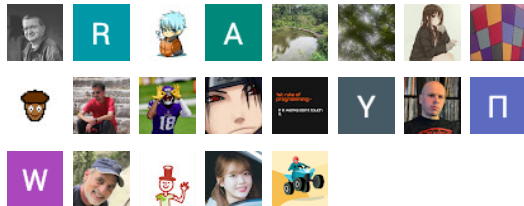
- ▶ [2025](#) (4)
- ▶ [2024](#) (9)
- ▶ [2023](#) (7)
- ▶ [2022](#) (18)
- ▶ [2021](#) (17)
- ▶ [2020](#) (10)
- ▶ [2019](#) (6)
- ▶ [2018](#) (6)
- ▶ [2017](#) (12)
- ▶ [2016](#) (11)
- ▶ [2015](#) (17)
- ▶ [2014](#) (18)

- ▶ [2013](#) (26)
- ▼ [2012](#) (30)
 - ▶ [December](#) (2)
 - ▶ [October](#) (4)
 - ▶ [September](#) (4)
 - ▶ [August](#) (2)
 - ▶ [July](#) (6)
 - ▼ [June](#) (2)
 - [GADTs](#)
 - [Why free monads matter](#)
 - ▶ [May](#) (3)
 - ▶ [March](#) (1)
 - ▶ [February](#) (1)
 - ▶ [January](#) (5)

▶ [2011](#) (1)

Followers

Followers (395) [Next](#)



[Follow](#)

This work is licensed under a [Creative Commons Attribution 4.0 International License](#).
Simple theme. Powered by [Blogger](#).