A Neighborhood of Infinity

Monday, August 07, 2006

You Could Have Invented Monads! (And Maybe You Already Have.)

If you hadn't guessed, this is about monads as they appear in pure functional programming languages like Haskell. They are closely related to the monads of category theory, but are not exactly the same because Haskell doesn't enforce the identities satisfied by categorical monads.

Writing introductions to monads seems to have developed into an industry. There's a gentle Introduction, a Haskell Programmer's introduction with the advice "Don't Panic", an introduction for the "Working Haskell Programmer" and countless others that introduce monads as everything from a type of functor to a type of space suit.

But all of these introduce monads as something esoteric in need of explanation. But what I want to argue is that they aren't esoteric at all. In fact, faced with various problems in functional programming you would have been led, inexorably, to certain solutions, all of which are examples of monads. In fact, I hope to get you to invent them now if you haven't already. It's then a small step to notice that all of these solutions are in fact the same solution in disguise. And after reading this, you might be in a better position to understand other documents on monads because you'll recognise everything you see as something you've already invented.

Many of the problems that monads try to solve are related to the issue of side effects. So we'll start with them. (Note that monads let you do more than handle side-effects, in particular many types of container object can be viewed as monads. Some of the introductions to monads find it hard to reconcile these two different uses of monads and concentrate on just one or the other.)

Side Effects: Debugging Pure Functions

In an imperative programming language such as C++, functions behave nothing like the functions of mathematics. For example, suppose we

have a C++ function that takes a single floating point argument and returns a floating point result. Superficially it might seem a little like a mathematical function mapping reals to reals, but a C++ function can do more than just return a number that depends on its arguments. It can read and write the values of global variables as well as writing output to the screen and receiving input from the user. In a pure functional language, however, a function can only read what is supplied to it in its arguments and the only way it can have an effect on the world is through the values it returns.

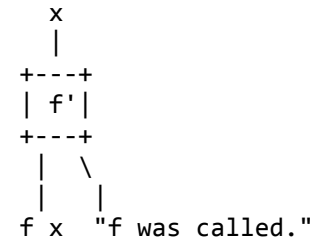
So consider this problem in a pure functional language: we have functions f and g that both map floats to floats, but we'd like to modify these functions to also output strings for debugging purposes. In Haskell, f and g might have types given by

```
f,g :: Float -> Float
```

How can we modify the types of f and g to admit side effects? Well there really isn't any choice at all. If we'd like f' and g' to produce strings as well as floating point numbers as output, then the only possible way is for these strings to be returned alongside the floating point numbers. In other words, we need f' and g' to be of type

```
f',g' :: Float -> (Float,String)
```

We can draw this diagrammatically as



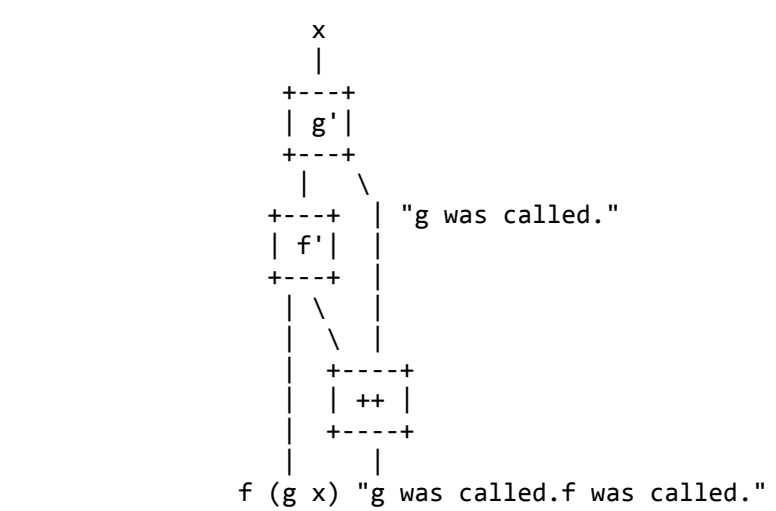
We can think of these as 'debuggable' functions.

But suppose now that we'd like to debug the composition of two such functions. We could simply compose our original functions, f and g, to form `f . g`. But our debuggable functions aren't quite so straightforward to deal with. We'd like the strings returned by f' and g' to be concatenated into one longer debugging string (the one from f' after the one from g'). But we can't simply compose f' and g' because the return

value of g' is not of the same type as the argument to f'. We could write code in a style like this:

```
let (y,s) = g' x
    (z,t) = f' y in (z,s++t)
```

Here's how it looks diagramatically:



This is hard work every time we need to compose two functions and if we had to do implement this kind of plumbing all the way through our code it would be a pain. What we need is to define a higher order function to perform this plumbing for us. As the problem is that the output from g' can't simply be plugged into the input to f', we need to 'upgrade' f'. So we introduce a function, 'bind', to do this. In other words we'd like

```
bind f' :: (Float,String) -> (Float,String)
```

which implies that

```
bind :: (Float -> (Float,String)) -> ((Float,String) -> (Float,String))
```

bind must serve two purposes: it must (1) apply f' to the correct part of g' x and (2) concatenate the string returned by g' with the string returned by f'.

Exercise One

Write the function bind.

Solution

```
bind f' (gx,gs) = let (fx,fs) = f' gx in (fx,gs++fs)
```

Given a pair of debuggable functions, f' and g', we can now compose them together to make a new debuggable function `bind f' . g'`. Write this composition as `f'*g'`. Even though the output of g' is incompatible with the input of f' we still have a nice easy way to concatenate their operations. And this suggests another question: is there an 'identity' debuggable function. The ordinary identity has these properties: `f . id = f` and `id . f = f`. So we're looking for a debuggable function, call it unit, such that `unit * f = f * unit = f`. Obviously we'd expect it to produce the empty debugging string and otherwise act a bit like the identity.

Exercise Two

Define unit.

Solution

```
unit x = (x,"")
```

The unit allows us to 'lift' any function into a debuggable one. In fact, define

```
lift f x = (f x,"")
```

or more simply, `lift f = unit . f`. The lifted version does much the same as the original function and, quite reasonably, it produces the empty string as a side effect.

Exercise Three

Show that `lift f * lift g = lift (f.g)`

In summary: the functions, bind and unit, allow us to compose debuggable functions in a straightforward way, and compose ordinary functions with debuggable functions in a natural way.

Believe it or not, by carrying out those two exercises you have defined your first monad. At this point it's probably not clear which of the structures we've looked at is the monad itself, or what other monads might look like. But rather than defining monads now I'll get you to do some more easy exercises that will introduce other monads so that you'll see for yourself that there is a common structure deserving of its own name. I'm also pretty confident that most people, faced with the original problem, would eventually have come up with the function bind as a way to glue their debuggable functions together. So I'm sure that you could have invented this monad, even if you didn't realise it was a monad.

A Container: Multivalued Functions

Consider the the functions sqrt and cbrt that compute the square root and cube root, respectively, of a real number. These are straightforward functions of type `Float -> Float` (although sqrt will thrown an exception for negative arguments, something we'll ignore).

Now consider a version of these functions that works with complex numbers. Every complex number, besides zero, has two square roots. Similarly, every non-zero complex number has three cube roots. So we'd like sqrt' and cbrt' to return lists of values. In other words, we'd like

```
sqrt',cbrt' :: Complex Float -> [Complex Float]
```

We'll call these 'multivalued' functions.

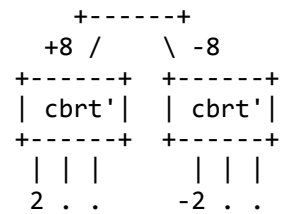
Suppose we want to find the sixth root of a real number. We can just concatenate the cube root and square root functions. In other words we can define sixthroot x = sqrt (cbrt x).

But how do we define a function that finds all six sixth roots of a complex number using sqrt' and cbrt'. We can't simply concatenate these functions. What we'd like is to first compute the cube roots of a number, then find the square roots of all of these numbers in turn, combining together the results into one long list. What we need is a function, called bind say, to compose these functions, with declaration

```
bind :: (Complex Double -> [Complex Double]) -> ([Complex Double] -> [Complex Double])
```

Here's a diagram showing how the whole process looks. We only want to write cbrt' once but still have it applied to both sqrt' values.

```
      64
      |
   +------+
   + sqrt'+
```



Exercise Four

Write an implementation of bind.

Solution

```
bind f x = concat (map f x)
```

How do we write the identity function in multivalued form? The identity returns one argument, so a multivalued version should return a list of length one. Call this function unit.

Exercise Five

Define unit.

Solution

```
unit x = [x]
```

Again, define `f * g = bind f . g` and `lift f = unit . f`. lift does exactly what you might expect. It turns an ordinary function into a multivalued one in the obvious way.

Exercise Six

Show that `f * unit = unit * f = f` and `lift f * lift g = lift (f.g)`

Again, given the original problem, we are led inexorably towards this bind function.

If you managed those exercises then you've defined your second monad. You may be beginning to see a pattern develop. It's curious that these entirely different looking problems have led to similar looking constructions.

A more complex side effect: Random Numbers

The Haskell random function looks like this

```
random :: StdGen → (a,StdGen)
```

The idea is that in order to generate a random number you need a seed, and after you've generated the number you need to update the seed to a new value. In a non-pure language the seed can be a global variable so the user doesn't need to deal with it explicitly. But in a pure language the seed needs to be passed in and out explicitly - and that's what the signature of random describes. Note that this is similar to the debugging case above because we are returning extra data by using a pair. But this time we're passing in extra data too.

So a function that is conceptually a randomised function `a → b` can be written as a function `a → StdGen -> (b,StdGen)` where StdGen is the type of the seed.

We now must work out how to compose two randomised functions, f and g. The first element of the pair that f returns needs to be passed in as an input to g. But the seed returned from the g also needs to be passed in to f. Meanwhile the 'real' return value of g needs to be passed in as the first argument of f. So we can give this signature for bind:

```
bind :: (a → StdGen → (b,StdGen)) → (StdGen → (a,StdGen)) → (StdGen → (b,StdGen))
```

Exercise Seven

Implement bind

Solution

```
bind f x seed = let (x',seed') = x seed in f x' seed'
```

Now we need to find the 'identity' randomised function. This needs to be of type

```
unit :: a → (StdGen → (a,StdGen))
```

and should leave the seed unmodified.

Exercise Eight

Implement unit.

Solution

```
unit x g = (x,g)
```

or just

```
unit = (,)
```

Yet again, define `f * g = bind f . g` and `lift f = unit . f`. lift does exactly what you might expect - it turns an ordinary function into a randomised one that leaves the seed unchanged.

Exercise Nine

Show that `f * unit = unit * f = f` and `lift f * lift g = lift (f.g)`

Monads

It's now time to step back and discern the common structure.

Define

```
type Debuggable a = (a,String)
type Multivalued a = [a]
type Randomised a = StdGen → (a,StdGen)
```

Use the variable m to represent Debuggable, Multivalued or Randomised. In each case we are faced with the same problem. We're given a function `a -> m b` but we need to somehow apply this function to an object of type m a instead of one of type a. In each case we do so by defining a function called bind of type `(a → m b) -> (m a → m b)` and introducing a kind of identity function `unit :: a → m a`. In addition, we found that these identities held: `f * unit = unit * f = f` and `lift f * lift g = lift (f.g)`, where '*' and lift where defined in terms of unit and bind.

So now I can reveal what a monad is. The triple of objects `(m,unit,bind)` is the monad, and to be a monad they must satisfy a bunch of laws such as the ones you've been proving. And I think that in each of the three

cases you'd have eventually come up with a function like bind, even if you might not have immediately noticed that all three cases shared a common structure.

So now I need to make some contact with the usual definition of Haskell monads. The first thing is that I've flipped the definition of bind and written it as the word 'bind' whereas it's normally written as the operator >>=. So bind f x is normally written as `x >>= f`. Secondly, unit is usually called return. And thirdly, in order to overload the names >>= and return, we need to make use of type classes. In Haskell, Debuggable is the Writer monad, Multivalued is the List monad and Randomised is the State monad. If you check the definitions of these

Control.Monad.Writer Control.Monad.List Control.Monad.State

you'll see that apart from some syntactic fluff they are essentially the definitions you wrote for the exercises above. Debugging used the Writer monad, the multivalued functions used the List monad and the random number generator used the State monad. You could have invented monads!

Monad Syntax

I don't want to spend too long on this (and you can skip this section) because there are plenty of excellent introductions out there.

You've already seen how the bind function can provide a nice way to plumb functions together to save you writing quite a bit of ugly code. Haskell goes one step further, you don't even have to explicitly use the bind function, you can ask Haskell to insert it into your code automatically.

Let's go back to the original debugging example except we'll now use the official Haskell type classes. Where we previously used a pair like `(a,s)` we now use `Writer (a,s)` of type `Writer Char`. And to get the pair back out of one of these objects we use the function runWriter. Suppose we want to increment, double and then decrement 7, at each stage logging what we have done. We can write

```
return 7 >>= (\x -> Writer (x+1,"inc."))
    >>= (\x -> Writer (2*x,"double."))
    >>= (\x -> Writer (x-1,"dec."))
```

If we apply runWriter to this we should get `(15,"inc.double.dec.")`. But

it's still pretty ugly. Instead we can use Haskell do syntax. The idea is that

```
do x <- y
    more code
```

is automatically rewritten by the compiler to

```
y >>= (\x -> do
                more code).
```

Similarly,

```
do
    let x = y
    more code
```

is rewritten as

```
(\x -> do
        more code) y
```

and

```
do
    expression
```

is just left as the expression.

We can now rewrite the code above as:

```
do
    let x = 7
    y <- Writer (x+1,"inc\n")
    z <- Writer (2*y,"double\n")
    Writer (z-1,"dec\n")
```

The notation is very suggestive. When we write y <- ... it's as if we can pretend that the expression on the right hand side is just x+1 and that the side-effect just looks after itself.

Another example. Write our sixth root example in the cumbersome form:

```
return 64 >>= (\x -> sqrt' x) >>= (\y -> cbrt' y)
```

We can now rewrite this as the readable

```
do
    let x = 64
    y <- sqrt' x
    z <- cbrt' y
    return z
```

We're able to write what looks like ordinary non-multivalued code and the implicit bind functions that Haskell inserts automatically make it multivalued.

Inventing this syntax was a work of genius. Maybe you could have invented it, but I'm sure I wouldn't have. But this is extra stuff is really just syntactic sugar on top of monads. I still claim that you could have invented the underlying monads.

Input/Output

There's now one last thing we have to look at before you're fully qualified in monadicity. Interaction with the outside world. Up until now everything I have said might apply to any pure functional language. But now consider lazy pure functional languages. In such a language you have no idea what order things will be evaluated in. So if you have a function to print the message "Input a number" and another function to input the number, you might not be able to guarantee that the message is printed before the input is requested! Go back to the randomised function example. Notice how the random seed gets threaded through your functions so that it can be used each time random is called. There is a kind of ordering going on. Suppose we have `x >>= f >>= g`. Because g uses the seed returned by f, we know for sure that f will generate its random number before g. This shows that in principle, monads can be used to order computations.

Now consider the implementation of random in the compiler. It's typically a C or assembler routine linked into the final Haskell executable. If this routine were modified to perform I/O we could guarantee that the I/O in f was performed before that in g. This is exactly how I/O works in Haskell, we perform all of the I/O in a monad. In this case, a function that conceptually is of type `a -> b`, but also has a side-effect in the real world, is actually of type `a -> IO b`. Type `IO` type is a black box, we don't need to know what's in it. (Maybe it works just like the random example, maybe not.) We just need to know that `x >>= f >>= g` performs the I/O in f before that in g.

Category Theory

One last thing. Monads were originally developed in the context of category theory. I'll leave the connection for another day.

Oh...and I ought to mention...I'm still not convinced that I could have invented Spectral Sequences. But I'm still working on it thanks to Tim Chow.

Appendix: A complete code example using random numbers

Firstly, here's the code corresponding to the definitions above:

```
import Random

bind :: (a -> StdGen -> (b,StdGen)) -> (StdGen -> (a,StdGen)) -> (StdGen -> (b,StdGen))
bind f x seed = let (x',seed') = x seed in f x' seed'
unit x g = (x,g)
lift f = unit . f
```

So here's what we're going to do: we want to construct a 2 (decimal) digit random number in three steps. Starting with zero we:
(1) add a random integer in the range [0,9]
(2) multiply it by 10
(3) add another random integer in the range [0,9]

Conceptually this operation is a composition something like this `addDigit . (*10) . addDigit`. But we know we need to thread the random seed through this code. Firstly, look at the actual definition of addDigit:

```
addDigit n g = let (a,g') = random g in (n + a `mod` 10,g')
```

This returns a pair consisting of n+random digit and the new seed. Note how it also takes that extra seed as argument. One could argue that 'conceptually' this function is `addDigit n = let a = random in n + a `mod` 10` and in a suitable strict and impure language that might be considered valid code.

Now consider the operation to multiply by 10. This is an ordinary function that we can 'upgrade' using lift. So we get

```
shift = lift (*10)
```

And now is the crucial bit. We can't simply compose but must instead use bind to convert our functions into a form that can be composed. So that conceptual `addDigit . (*10) . addDigit` becomes

```
test :: Integer -> StdGen ->  (Integer,StdGen)
test = bind addDigit . bind shift . addDigit
```

Now we create a seed using my favourite number and run the code:

```
g = mkStdGen 666
main = print $ test 0 g
```

Note that I've chosen not to use the Monad type class so that nothing is hidden from you and everything is explicit.

Update: Thanks for the comments. I've now made lots of fixes and added an appendix with a random number example. Most of the code fragments have now been copied back from the final text into real code so there should only be a small number of typos left. Apologies for any confusion caused!

For a slightly different take on monads I've written a little on Kleisli arrows. And if you're interesting in layering monads to achieve more functionality you can try this.

Posted by Dan Piponi at Monday, August 07, 2006
Labels: haskell

65 comments:

Korollary said...

Nice tutorial. I think exercise 1's solution should end with "(fx,gs++fs)", though.

Also, in the following paragraph 'bind' should be in monospace in bind f' . g'.

[nitpick]In the paragraph that explains Haskell's conventions for the terms, there's no reason to say that unit is 'usually' called return; it's been officially return for the last eight years.[/nitpick]

This tutorial reminded me of Wadler's original monads paper. I like it. It doesn't get simpler than this, and it may still not be possible to wrap one's head around it just by reading it. But again, how much math can you learn without solving any exercises?

Tuesday, 08 August, 2006

kowey said...

Very nice! (Now if only you could do the same for arrows :-D)

Wednesday, 09 August, 2006

sigfpe said...

korollary,

Yes, gs++fs. How many times can I check something knowing precisely the error I'm likely to make and still get it wrong?!

Thanks for the corrections. I had a really hard time managing the fonts and the layout as this web site uses slightly different conventions to regular HTML.

Wednesday, 09 August, 2006

sigfpe said...

kowey,

Actually, I have a document on Arrows sitting in my unpublished drafts section on blogger.com.

'Arrows' is a really stupid name because in saying what an arrow is I'm forced to use the word 'arrow' in two diferent senses, one for Hughes meaning and one for the usual meaning from category theory. It's like deciding that the prime numbers bigger than 20 are interesting for some reason and then choosing to give them a name, 'the primes'. I'll use capital 'A' for Hughes' sense.

Arrows really are just arrows in a certain type of category. It can get fiddly to pin down exactly what the 'certain type' is - but that's partly because the original paper doesn't pin it down exactly. But arrows in many familiar categories like **Set** are Arrows and you can get a very accurate intuition about Arrows simply by thinking that an Arrow from A to B behaves much the same as a function from A to B. (Eg. you can compose them, form arrows to products and coproducts and so on.)

In a computational context I don't think Arrows give you much at all. I think that the main application I've seen, functional reactive programming, works much better with comonads than with Arrows, but that's just my opinion. When I realised this I decided not to publish my Arrows document after all.

Hughes did propose a nice syntactic sugar for Arrows and that might make them useful for some applications. In fact, maybe the sugar is the point.

Anyway, I haven't answered your request, just given an excuse for not doing it!

Wednesday, 09 August, 2006

Chris Bogart said...

I've had some lingering confusion about monads, so I've been working through this. It's great so far, but I've been banging my head against exercise 7 for a few hours, and I'm honestly not sure whether you have a typo or if I'm just hopelessly confused.

First of all, it seems to me that the result of binding a randomized function of type
a -> StdGen -> (b, StdGen)
should be something like
(a -> StdGen -> (b, Stdgen)) -> (a, StdGen) -> (b, StdGen)

Not sure about the a's and b's, but since G is going to return a pair, bind-f should take a pair and return a pair. Right?

Also, I think you have an impossibility in your text when you say that the first element of what f returns gets passed to g, but the seed returned from g gets passed to f.

Finally, seems like the answer to the exercise should be:

bind f (x, seed) = f x seed

Your answer seems to have a **let** in it that pairs up a couple arguments then unpairs them to no effect. However, I'm just learning Haskell, so if you assure me what you have is correct, I'll keep playing with it till I understand it:-)

Thanks for the tutorial!

Chris

Wednesday, 09 August, 2006

Chris Bogart said...

Bah -- never mind, I figured it out. Although you're sequencing a couple functions (say, f and g), the magical goodness that gets passed between them is not just the pair (b,StdGen), but a whole function, namely g with the first argument curried. This makes my brain hurt, but in a good way.

I still think the function names are mixed up once in the text, but I see now that the bind type and answer make sense. Also, the parameters x and x' in the answer to exercise 7 do not have the same relationship as seed and seed' -- that caused me to miss that x was a function.

Thanks,
Chris

Wednesday, 09 August, 2006

sigfpe said...

chris,

There may be something screwy in the last example. I'll go over it again.

Thursday, 10 August, 2006

Fritz Ruehr said...

A little typo, I think: about half way through the tutorial is this bit:

... and introducing a kind of identity function bind :: a → m a.

This should be "unit :: a → m a", right?

Very nice tutorial, by the way.

Monday, 14 August, 2006

sigfpe said...

Thanks Fritz. I'll fix that when I get a chance...

Monday, 14 August, 2006

Jim said...

Very nice tutorial by the way.. It's the first i was able to get my brother to read thru without hearing any grumbling :)

However there is an error in your do notation:

```
do
let x = 7
y <- Writer (x+1,"inc\n")
z <- Writer (2*x,"double\n"))
Writer (x-1,"dec\n")
```

yields 6 and is thus not equivalent to

```
7 >>= (\x -> Writer (x+1,"inc."))
>>= (\x -> Writer (2*x,"double."))
>>= (\x -> Writer (x-1,"dec."))
```

which yields 15... (Basicly you made bindings for y and z in the do notation, but went on to do nothing with them.)

Thursday, 17 August, 2006

sigfpe said...

Thanks Jim. I'll fix that soon. I must learn not to tweak code for aesthetic reasons (to get x, y and z in alphabetical order) without testing it!

Thursday, 17 August, 2006

Josef said...

[*Blogger doesn't allow any tag that gives fixed width text so I've made all code bold instead. But it doesn't help me indent things properly :-(*]
Others have already pointed out some typos in your excellent tutorial and I hate to say it but I've found some more. The last two examples on your section on "Monad Syntax" are just wrong. They're so wrong that I'm not even sure what the right version should be. Anyway, here's the first problematic one:

64 >>= (\x -> sqrt' x) >>= (\y -> cbrt' y)

First of all the parenthesis are both wrong and unnecessary. Secondly, **64** is not a monadic computation. So a correct version could be the following:

return 64 >>= \x -> sqrt' x >>= \y -> cbrt' y

or

let x = 64 in sqrt' x >>= \y -> cbrt' y

or even

sqrt' 64 >>= \y -> cbrt' y

The last example in the section is also wrong. The last expression in the **do** block, the variable **y**, is not a monadic computation. Furthermore, the variable names does not match the previous examples. Here's what one could write instead (with apologies for the indentation):

```
do let x = 64
y <- sqrt' x
cbrt' y
```

HTH.

Sunday, 27 August, 2006

sigfpe said...

Thanks Josef.

I should have just written in literate Haskell so that the whole thing compiled rather than copying and pasting piecemeal from bits of code and then editing them by hand afterwards. Very shoddy of me!

Sunday, 27 August, 2006

weierstrass said...

i've been really loving the articles you've written recently, on this and related stuff; the one about 'area' and euler characteristics was fascinating.

but for some reason this article has been reappearing over and over again in my rss feed (i subscribe to your blog using bloglines). i don;t know whether this is something you're doing , or a blogger / bloglines problem, but i just thought i'd let you know in case you able to fix it.

keep up the good work.

Monday, 28 August, 2006

archiver said...

g f x "g was called.f was called."

shouln't this line be as below?

f g x "g was called.f was called."

Sunday, 10 September, 2006

sigfpe said...

archiver,

Glad to see readers are keeping on their toes! :-)

Monday, 11 September, 2006

Shane said...

I read this article a couple days ago. I continued reading Hudak's book, The Haskell School of Expression, and came upon his chapter on monads. Having read this article really prepped my mind for that chapter. As I read the chapter I kept referring back in my mind to your ideas 'unit' and 'bind', which really helped. I'm really excited about being able to grasp and finally exercise monads. Thanks for a great article.

Tuesday, 19 September, 2006

Anonymous said...

This is the only useful tutorial on monads that ever existed for a newbie. Do not think otherwise and belive me! Many thanks! At least now I have a comfort and calm soul! There are bunch of tutorials there that say how these monadic thing are clever and fantastic. For a long time I was wandering along and between them. I only can say Thank you very much!

Tuesday, 17 October, 2006

Vincent said...

it would seem to me there's a typo in diagram 2, i.e. the one following the text "Here's how it looks diagramatically:".

the result of the function application should in my opinion be $f\ g\ x$ rather than $g\ f\ x$, since you first apply g' to x, followed by the application of f to the result. but perhaps i'm just misinterpreting your notation?

Thursday, 11 January, 2007

Adam said...

Thanks for explaining this better than any other tutorial I've found.

Thursday, 01 March, 2007

tikitu said...

Great tutorial, thanks! Something odd about the leadup to exercise 7 though, I don't think this can be right:

The first element of the pair that f returns needs to be passed in as an input to g. ... Meanwhile the 'real' return value of g needs to be passed in as the first argument of f.

Thursday, 22 March, 2007

sigfpe said...

tikitu,

Maybe this weekend will be the time when I finally get around to fixing the slight problem you've spotted...

Thursday, 22 March, 2007

sigfpe said...

A few people have reported problems with this post, eg. the comment just above. I can't see the problem but enough people have complained that I've a hunch there is at least a small type in some part of the code somewhere. A virtual beer for anyone who can find it.

Sunday, 25 March, 2007

tikitu said...

I'll take that challenge! Actually I don't think there's anything wrong with the code (ok, I didn't actually try it all, but it seems sensible). I think in the sentence I picked on there's confusion about whether you're composing f with g or g with f.

Assuming we want f (g x), I think it should read something like (modulo formatting):

We now must work out how to compose two randomised functions, f and g. The seed returned from g needs to be passed in to f. But the 'real' return value of g also needs to be passed in as the first argument of f. So we can give this signature for bind:

(Btw, after sitting up half the night with your tutorial I tried again in the daytime, and everything was illuminated!)

Thursday, 29 March, 2007

Davor said...

I think your solutions to exercises 1 and 4 don't match the signatures you gave when introducing the exercise. For example, the signature for

```
bind f (gx, gs) =
 let
  (fx, fs) = f gx
 in
  (fx, gs ++ fs)
```

is

```
bind :: (Double -> (Double, String)) -> (Double, String) -> (Double->String)
```

Instead, should bind return a function?

```
bind f = \(gx, gs) ->
 let
  (fx, fs) = f gx
 in
  (fx, gs ++ fs)
```

And similarly for exercise four.

Monday, 23 April, 2007

Davor said...

Nevermind my previous comment, it was off base. (Because of currying, I think.)

Monday, 23 April, 2007

sigfpe said...

Davor,

Those two implementations of bind are more or less equivalent. The type a -> b -> c can be thought of in two ways: as a function taking two arguments, one of type a and one of type b, and returning an instance of type c, but also as a function that takes an instance of a as argument and returns a function of type b -> c. Haskell makes no distinction because every function is Curried.

A simple example:

```
g :: Integer -> Integer -> Integer
g x y = x+y
```

and

```
g' :: Integer -> Integer -> Integer
g' x = \y -> x+y
```

define more or less identical functions. Test them out interactively in hugs or ghci. Both g 0 and g' 0 are functions of type Integer -> Integer.

Monday, 23 April, 2007

Anonymous said...

I hope you also write up a " You Could Have Invented Applicative Functors! (And Maybe You Already Have.)"

Sunday, 03 June, 2007

Anonymous said...

Finally an understandable introduction to Monads (have already read a few of them), that still gets to the point: After a few examples, I really really appreciate reading a concise definition, like in this case that a monad IS a certain kind of triple...

Thank you!

Thursday, 12 July, 2007

Pupeno said...

Hello,

Excellent article! It was so good that I started it reading once... then stopped using Haskell and about a year latter, when I am playing again with Haskell I still reminded it and I just came back to read it.

Thank you for writting it and for posting it.

Now, a little suggestion, I believe that the piece of code just before exercise two would be more understandable if it said "unit * f' = f' * unit".

Friday, 13 July, 2007

JY said...

I'm confused. In your definition, "bind" has the type (a → m b) -> m a → m b. But in Haskell, bind (>>=) has the type m a -> (a -> m b) -> m b... Do I miss something?

Monday, 20 August, 2007

sigfpe said...

JY,

bind is just >>= with the arguments reversed. I chose the way round that I did to make clearer the analogy between function composition and monadic composition.

With bind you write the function on the left of the arguments, just like with function composition. With >>= the function goes on the right, hiding the similarity with function composition.

Monday, 20 August, 2007

David said...

Misleading wording in the paragraph already fingered above aside, this text is awesome! Honest thanks, you totally demystified monads to another HS newbie. And that's coming from a social sci major :) .

Thursday, 13 September, 2007

Anonymous said...

Can you propose asolution in Haskell for the more general problem of computing the input of a function from its result?
This is the general problem of solving equations, and we know that there may be several solutions to the problem, that the simple concept of currying will not solve always, or that will solve returning multiple solutions.
The concept could however be modeled by stating that the function input can be solved as a list that may be empty.
Is there a way to extend the concept of currrying a function one step further so that functions are now treated only as a pair consisting of its input and its output, so that we can pass it either the input or the output and compute the missing element of the pair according to the function definition?
Under this interpretation, we would not make any semantic difference between the function inputs (whatever their number) and the function outputs (also whatever their numbers): they are just indexed elements of a tuple, and currying is just one way to model the association (although currying just favors only one direction of solving problem, from the first member of the tuple to the last member).

My idea is to be able to represent also, for example a database table (with unique rows) as a function, and each row in the table is a tuple of elements that can be either interpreted as inputs or outputs of a function, so that we can solve the problem of finding the value of a missing column by specifying the value of the other columns.

To model a database table with non unique rows, we could reuse the same, because every row in a database table also has an implicit supplementary column that makes the row unique: the row number. under this representation, the database table canbe viewed as a curried function that takes a row number and returns the matching row as a tuple. But it could also take the tuple as input and return a list of matching row numbers. This is what I meanin a generalized currying concept: not favoring any direction of resolution. And here the concept of Monads could help by allowing to get in the return of a problem the list of values we are looking for in its return, "monadized" with the rest of the other column values given to the problem, the monad itself representing the association.

Let's think about the type of such function:
* a table is now in fact an association whose type is a list of tuples of several types, one type for each column.
* to solve a problem with this table, we specify it also with a tuple (or list of tuples) filled with the values of the column we know, and the function matches it by binding the columns in the input list of tuples with its own internal list of

tuples modeling each row in the table. When a match is found, the function returns this row in the list of inputs.
* now let's look at the function that takes a numeric input and return the input plus 1: its is similar as it can be viewed as anassociation of tuples of the form (x, x+1) so the function or table is modeled as a list of pairs. The problem it can solve would not only be the forward problem of finding a matching association (x,x+1) when we know the value of the first column x as a list of matching values for x+1, but also finding solutions for the reverse problem.
For forward problems, I want to be able to give to the function that internally stores the association as list of pairs, i.e. [(int,int)], feed it with a tuple of lists of type ([int],unknown) which represent the values we know, the function will perform the internal resolution by binding this input with its internal list of tuples and will return a tuple of lists filled with the values of the "unknow" elements.
so conceptually, the function I need is now of "type":
([int],unkown)->(???,[int])
but could also return just the list of its rows that match the input:
([int],unkown)->[(???,int)]
As there's no direction favored, the generalization of the problem will required using the more general problem of representing the problems we can solve with this function as:
[(int|unknown,int|unknown)]->[(int|unknown,int|unknown)]
in other words, the type of a problem with its solution according to a function is a pair of types, whose signature are both equals the type signature of the function itself!
Regading the function "+1" above, it is just a particular solution to the geneal concept represented in [(x,x+1)] as if it was a database table containing all the associations (as as list of tuples) between is input and its output.

Friday, 05 October, 2007

Anonymous said...

Now let's say how I want the modeled function behave.

I'll take the example of the list of associations of the form (x,x+1) which I'll name "inc".

The associations that "inc" models are of type (int,int) but to make the problem solvable, it internally includes a supplementary leading element in the tuple, so it is in fact internally:
(id,[(int,int)])
with an additional type representing the "id"entity of a solution. If this identity is known and is a solution to the "inc" function, then the list which which it is associated contains a single pair of type (int,int) and its value (x,y) satisfies the condition y=x+1.

When coding the function, I want to be able to store it as a list of binding conditions representing the associations as a binding condition defining the satisfied association precisely as:

[(id,x,x+1)]

(the effective value of the id does not matter, it should be hidden by some

syntax sugar of the language)

and the function will also encode several solutions to allow the problems to be solved according to the values of the variables :
[(id,x,x+1)]
[(id,y-1,y)]

(here also the value of the id should not be needed for the problems we need to solve, as the compiled function will internally compute them.

So instead of encoding in the syntax the places where the id is present, we will assume that the syntax provides such sugar to do that, so that the function is can be written by just specifying in its definition:
[(x, x+1)]
OR
[(y-1, y)]

In other words "inc" is now defined precisely as a list of tuples that it considers equivalent:
[(x, x+1), (y-1, y)]

When "running" the function, we'll give similar list of tuples in input, the function will try to bind each tuple of its input with each tuple present in the list of its own definition, and if a match is found, it will return this matching tuple with the variables bounds in the list of tuples that are part of the solution.

For making matches, the internal binding made by the "running" function will look for free variables present in its internal list of solutions, so given the input [(z,4)] that contains a free variable z for our problem, it binds successively each tuple of the input, i.e. (z,4) with each tuple in the list of proposed solutions:

* the first solution is the tuple (x,x+1), so binding it with the input tuple (z,4) would require binding x with z (this is possible) and x+1 with 4 (i.e. z+1 with 4, but this won't work because you can't bind an expression containing free variables from the input to a constant that has no free variable); this first solution is then eliminated from the output

* the second solution proposed by "inc" is the tuple (y-1,y), and binding it with the input (z,4) is possible: y-1 is bound to z (possible because both expressions contain free variables, this is the same as an equation y-1=z) and the second element y of the solution is bound to 4, so now y=4, and the tuple (y-1=z,y=4) is part of the solution, this tuple can be reduced because now y is not free and can be eliminated: (4-1=z,4) so finally the solution is (3=z,4)

Finally the function returns the list of tuples [(3=z,4)] for the input [(z,4)], and this is exactly what we wanted to solve.

The language may need to eliminate parts that are equivalent between the input and output, to provide a more readable solution by just keeping "3=z", i.e. the solution to the problem where z was free is now binding z to a value according to the definition of the "inc" function to which it was given. So the final solution is [(3,4)]

In other words: inc [(z,4)] returns a [(3,4)] each solution being annotated with bindings (which are implicit here because they are part of the input/output parameters to the function in the same order, so the bining z=3 is implicit from the problem it had to solve)

The difference here is that we are not limited by the curryfication.

Such model will easily represent as a "function" any database table.

The function would be written effectively something like:

"inc" :: [(int,int)] -> [(int,int)]
return [(x,x+1)] union [(y-1,y)]

There are a few other missing syntaxic sugar here to make it work (notably, I used the term "list" everywhere when in fact the list is not ordered but is actually a "set" with no duplicate elements). This requires some more work to perform set operations (as suggested above when I introduced the word "union").

How can I represent this extended "function" in Haskel? Can a monad represent such thing?

Friday, 05 October, 2007

Jonathan said...

As an alternative to

```
> do
> let x = 7
> y <- Writer (x+1,"inc\n")
> z <- Writer (2*y,"double\n")
> Writer (z-1,"dec\n")
```

I believe you could also write

```
> do
> let x = 7
> x <- Writer (x+1,"inc\n")
> x <- Writer (2*x,"double\n")
> Writer (x-1,"dec\n")
```

Is that right?

Saturday, 20 October, 2007

sigfpe said...

Jonathan,

Yes, that's fine. On each line you're introducing a new x.

Saturday, 20 October, 2007

d said...

Hi, I think that I finally got this monad stuff. But there's a problem when I try to run the example from the appendix. Here's the error message:
Couldn't match expected type `(Integer, StdGen)'
against inferred type `Integer -> StdGen -> (Integer, StdGen)'
In the expression: (bind addDigit) . ((bind shift) . addDigit)

Wednesday, 14 November, 2007

d said...

OK, I've typed
test x g = (bind addDigit) . (bind shift) . addDigit
instead of
test = (bind addDigit) . (bind shift) . addDigit
Sorry for the interruption

Wednesday, 14 November, 2007

Andrew Haley said...

One thing that confused me horribly about this paper.

Sometimes you use "->" and sometimes "→", as in

(a → m b) -> (m a → m b)

I thought that there might be some difference.

Would it be too much to change this?

Tuesday, 08 January, 2008

sigfpe said...

Andrew,

Sorry that caused confusion. Just treat the two symbols the same way. At that point I was still trying to figure out which symbol was better to use and hadn't settled for one or the other. If I get time I'll try to make it consistent.

Tuesday, 08 January, 2008

knut said...

Great tutorial! I especially liked the comparison between binding and function composition. In fact, for your examples, I find the "f*g" notation more succinct than both using >>= directly and the do notation. So far I haven't found an equivalent operator in Haskell, so for now I'm using the following:

```
module MonadUtils where
(°) :: Monad m => (b -> m c) -> (a -> m b) -> (a -> m c)
f ° g = (>>= f) . g
```

While probably not suitable for IO, it's a pretty neat way of writing stuff like

sixthrt' = sqrt' ° cbrt'
(x',w) = runWriter $ (f ° g ° h) x

Friday, 01 February, 2008

sigfpe said...

knut,

The function you defined is called "composition of Kleisli arrows". I wrote a bit about it here: http://sigfpe.blogspot.com/2006/06/monads-kleisli-arrows-comonads-and.html

Friday, 01 February, 2008

Marijn Schouten said...

About the randomized functions: All the confusion is due to Curry-Howard iso making it unclear what is an argument and what is the return value.
Depending on your view you get two different monad types. When you say "The first element of the pair that f returns" it seems clear that you view f as a 2-argument function which returns a pair and thus that the monad type should be (a, StdGen), but your solution of bind is the one for monad type StdGen -> (a, StdGen) which results from viewing f as a function of one argument which returns a function.
Your choice of names x and x' in your definition of bind is VERY confusing. Might I suggest instead: bind f r = \s -> let (x, s') = r s in f x s'? Other suggested rewording:
"We now must work out how to compose two randomized functions, f and g. The pair that the function that is returned by g returns needs to be decomposed and passed in as input to f. So we can give this signature for bind:".
It would also be very good if you could explain what happens if you choose the monad type to be (a, StdGen) and why that is a bad idea (if it is).
Lastly, I do think you offer a very enlightening view of monads, so thank you.

Thursday, 05 June, 2008

Anonymous said...

In the line "Where we previously used a pair like (a,s) we now use Writer (a,s) of type Writer Char.", shouldn't it be type Writer [Char]?

Thanks.

Monday, 14 July, 2008

Anonymous said...

Nice tutorial, but I don't like "your favorite number" ;>

Thursday, 12 February, 2009

James Brooks said...

Great post, really made monads easy to understand.

Any chance you could write the same sort of thing on call/cc ?

Thursday, 06 August, 2009

Loup Vaillant said...

About the do notation, you basically said:

do {let x = y; more code} ==>> (\x -> do more code) y

I was confused at first, until I performed the obvious beta reduction:

let x = y in do more code

This is actually closer to the original do notation, closer to the other rules, and I think a bit clearer.

Anyway, thank you. I now have a way to explain monads to my poor C++ colleagues.

Tuesday, 24 November, 2009

DAY said...

It would be better to change unit * f = f * unit = f in the paragraph above Exercise Two into unit * f' = f' * unit = f'. And a space is missing in f'*g' in the same paragraph.

Thursday, 04 March, 2010

Andrew Whaley said...

Bravo - what a great tutorial ! I've read about a dozen articles and not really understood Monads but this one really hits the nail. Thanks.

Tuesday, 09 March, 2010

Anonymous said...

An excellent tutorial. It really helped me a lot in understanding monads. The key was the 'bind' function with the more natural order of parameters compared to '>>='. The '*' operator really made clear what is going on, and it was really simple to define it in terms of 'bind'. I also liked that you created exercises to force me stop and think, thus helping in the understanding of the concept.

On the downside: the wording before the bind function of the random number example needs a little bit of improvement, exercise 7 was the most difficult because of it.

Friday, 02 July, 2010

Anonymous said...

Is it an academic convention to write your type signatures like this?

bind :: (a → StdGen → (b,StdGen)) → (StdGen → (a,StdGen)) → (StdGen → (b,StdGen))

It would be much, much more clear if you wrote it like this:

bind :: (a → StdGen → (b,StdGen)) → (StdGen → (a,StdGen)) → StdGen → (b,StdGen)

Obviously the two are equivalent but it looks like you *intend* for the first one to return a function rather than accept a third argument.

Wednesday, 18 August, 2010

sigfpe said...

Anonymous,

The version I gave highlights a certain symmetry in the signature. It's of the form X -> F a -> F b. That's slightly hidden in your version. Of course they are equivalent, but it's this symmetry I want people to notice.

Thursday, 19 August, 2010

Anonymous said...

This tutorial is a great introduction to monads as it starts with the problem and let the reader develop (or even invent) the solution. This helped me a lot, thanks!

Wednesday, 17 November, 2010

Eternal said...

Thanks a lot for a great tutorial.

I've been searching for a while and this is the best one I've found so far, both in books and on the net.

Monday, 13 December, 2010

Ray Lee said...

For not being a Haskell programmer, I've read a fair number of explanations of monads over the past year (I blame Reddit). So far, I've hit two that have helped; an eight-word definition ("Monads provide a generalized interface to sequential computation."), and your exposition I just tripped across.

You've managed to take something that is somewhat impenetrable when presented by others and make the concept not merely natural but truly

inevitable once we reach a certain level of abstraction. I suspect this means you normally teach math.

Anyway, thank you for this.

Friday, 31 December, 2010

Anonymous said...

The best tutorial on Monads! - Thank you

Saturday, 11 February, 2012

Anonymous said...

I never comment on these things,
But thanks so much.

Thursday, 22 March, 2012

Tanneguy said...

Great, great article !
it is the only one I have read (and I read some) that defines a monad by what it *is* rather than by what it *does*.
I had been very frustrated and confused by the what-it-does approach of explaining monads... I even had been wondering if Haskell was not a complicated way to do simple things...

So thank you so much !

Wednesday, 09 May, 2012

Anonymous said...

Great Tutorial, thanks!

But I'm confused: In exercise 6 we are to prove that lift f * lift g = lift (f.g).

But, by definition, f . g is not defined (in contrast to exercise 3).

Sunday, 03 March, 2013

john said...

Brilliant stuff. This has really helped me. Thankyou !!!
This is a very useful tutorial which will never get outdated.
So please can the following be fixed for the sake and sanity of future readers (of which I'm sure there will be many)...

As DAY and and Pupeno have already said...

The piece of code just before exercise two would be more understandable if it said "unit * f' = f' * unit".

The original says "unit * f = f * unit = f" which is simply not true if "f" refers to the same f as mentioned previously in the article (which is what the ready would naturally assume).

This hurt my head for a couple of hours before I finally realised it was a mistake (or ambiguous if you prefer).

Thanks,
John

Thursday, 14 November, 2013

john said...

Brilliant stuff. This has really helped me. Thankyou !!!
This is a very useful tutorial which will never get outdated.
So please can the following be fixed for the sake and sanity of future readers (of which I'm sure there will be many)...

As DAY and and Pupeno have already said...

The piece of code just before exercise two would be more understandable if it said "unit * f' = f' * unit".

The original says "unit * f = f * unit = f" which is simply not true if f refers to the same f as mentioned previously in the article (which is what the ready would naturally assume).

This hurt my head for a couple of hours before I finally realised it was a mistake (or ambiguous at best).

Thanks,

Thursday, 14 November, 2013

AngryGamer said...

Confused with the sixth root example. You describe
sixth root = sqrt(cubrt x) yet the diagram shows cubrt (sqrt x).

Wednesday, 26 March, 2014

AngryGamer said...
This comment has been removed by the author.
Wednesday, 26 March, 2014

Sam Isaacson said...

For the Writer Monad (first example of debug state); why would you not just solve the problem with a function of the following type:

(Result, String) -> (Result, String)

The second parameter in the tuple can carry your logging info from function call to the next.

This appears simpler to a programmer without Monad experience.

Friday, 03 October, 2014

Post a Comment

Links to this post

Create a Link

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)

Blog Archive

Some Links

- The Comonad.Reader
- Rubrication
- Richard Borcherds: Mathematics and physics
- The n-Category Cafe
- Ars Mathematica

About Me

Dan Piponi

Blog: A Neighborhood of Infinity
Code: Github
Twitter: sigfpe
Home page: www.sigfpe.com

View my complete profile
