Bartosz Milewski's Programming Cafe

Category Theory, Haskell, Concurrency, C++

January 9, 2011

Monads for the Curious Programmer, Part 1

Posted by Bartosz Milewski under C++, Category Theory, Functional Programming, Haskell, Monads, Programming, Type System
[41] Comments

i
80 Votes

The Monad is like a bellows:
it is empty yet infinitely capable.
The more you use it, the more it produces;
the more you talk about it, the less you understand.

–Monad Te Ching

I don't know if I'm exaggerating but it seems like every programmer who *gets* monads posts a tutorial about them. (And each post begins with: There's already a lot of monad tutorials on the Internet, but…) The reason is that *getting* monads it's like a spiritual experience that you want to share with others.

When facing a monad, people often behave like the three blind men describing an elephant. You'll see monads described as containers and monads described as actions. Some people see them as a cover-up for side effects, others as examples of endofunctors in Category Theory.

Monads are hard to describe because they don't correspond to anything in our everyday experience. Compare this with Objects in Object-Oriented programming. Even an infant knows what an object is (something you can put in your mouth). What do you do with a monad?

But first, let me answer the pertinent question:

Why Bother?

Monads enable pure functional programmers to implement mutation, state, I/O, and a plethora of other things that are not functions. Well, you might say, they brought it on themselves. They tied their hands behind their backs and now they're bragging that they can type with their toes. Why should we pay attention?

The thing is, all those non-functional things that we are so used to doing in imperative programming are also sources of a lot of troubles. Take side effects for instance. Smart programmers (read: the ones who burnt their fingers too many times) try to minimize the use of global and static variables for fear of side effects. That's doable if you know what you're doing. But the real game changer is multithreading. Controlling the sharing of state between threads is not just good programming practice– it's a survival skill. Extreme programming models are in use that eliminate sharing altogether, like Erlang's full isolation of processes and its restriction of message passing to values.

Monads stake the ground between total anarchy of imperative languages and the rigid dictatorship of Erlang-like isolationism. They don't prohibit sharing or side effects but let you control them. And, since the control is exercised through the type system, a program that uses monads can be checked for correctness by the compiler. Considering how hard it it to test for data races in imperative programs, I think it's worth investing some time to learn monads.

There is also a completely different motivation: metaprogramming. The template language used for metaprogramming in C++ is a pure functional language (see my blog post, What does Haskell have to do with C++?). If monads are so important in functional programming, they must also pop up in C++ metaprogramming. And indeed they do. I hope to discuss this topic in a future post.

So what's a monad?

A Categorical Answer

If you don't know anything about category theory, don't get intimidated. This is really simple stuff and it will clarify a lot of things, not to mention earning you some bragging rights. My main goal is to share some intuitions from mathematics that will build foundations for a deeper understanding of monads in programming. In this installment I will explain categories, functors, and endofunctors, leading up to monads. I will give examples taken both from everyday life and from programming. I will really get into monads and their practical applications in the next installment, so be patient.

Categories

A *category* is a natural extension of our notion of sets and functions. The generalization of a set in a category is called an *object* (a pretty neutral term with little semantic ballast), and the generalization of a function is called a *morphism*. In fact, the standard example of a category is the category of sets and functions called (capital letter) **Set**.

A morphism (read "function") goes from one object (read "set") to another. Mathematical functions like *sin* or *exp* usually go from the set of real numbers to the set of real numbers. But you may also define functions like *isPrime* that go from natural numbers to Booleans, or a function *price* that goes from a set of goods to the set of numbers.

The only thing a mathematician needs to know about morphisms is that they can be composed. If you have a morphism from A to B, `A->B`, and another going from B to C, `B->C`, then they can be composed to a morphism from A to C, `A->C`. And just like the standard composition of functions, morphism composition must be associative, so we don't need parentheses when composing more than two of them.

Actually, two things. There must be, for every object, a special morphism called *identity* that essentially does nothing and when composed with any other morphism reproduces the same morphism.

Just to throw you off the track, a category doesn't have to be built on sets and functions. You can easily construct simple categories from blobs and arrows. Fig 1 shows such a category that contains two objects and four morphisms: arrows between them (formally, those arrows are ordered pairs of objects so, for instance, f is a pair (A, B)). You can easily check that any two morphisms can be composed and that the two moprphisms i_A and i_B serve as identities.

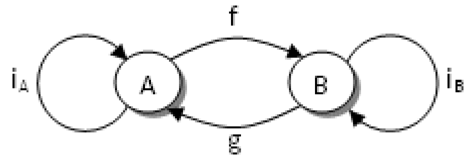
$$\begin{tikzcd} A \arrow[loop left, "i_A"] \arrow[r, bend left, "f"] & B \arrow[loop right, "i_B"] \arrow[l, bend left, "g"] \end{tikzcd}$$

Fig 1. A simple category with two objects and four morphisms.

That's it! Hopefully I have just convinced you that a category is not a big deal. But let's get down to Earth. The one category that's really important in programming languages is the category of types and functions, in particular its Haskell version called **Hask**. There usually is a finite set of basic types like integers or Booleans, and an infinite set of derived types, like lists of integers, functions from integers to Booleans, etc. In **Hask**, a type is just a set of values. For instance, the type `Char` is a set {'a', 'b', 'c', … }.

So, in the category **Hask**, types are *objects* and functions are *morphisms*. Indeed, a function maps one type into another (forget for a moment functions of multiple arguments– they can be modeled with currying– and polymorphic functions– they are families of functions). And these are functions in the functional-programming sense: called with the same values they return the same values–no side effects allowed.

Function composition is just passing the result of one function as an argument to another. The identity function takes x and immediately returns it back.

This is all fine, but what's in it for me, you might ask. So here's the first insight and a moment of Zen. If there is one thing that you can call the essence of programming, it's composability. In any style of programming you always compose your program from smaller pieces, and those pieces from even smaller pieces, and so on. That's why categories with their composable morphisms are so important. The essence of Lego blocks is the way they fit together, their composability, not the color or size. The essence of functional programming is how functions work together: how you can build larger functions from smaller ones.

Every category is defined by its choice of objects and morphisms. But is there something that can characterize a given category that's independent of its choice of particular objects and morphisms? How do you expose the inner structure of a particular category? Mathematicians know exactly how to do that. You have to be able to map categories into other categories while preserving some constraints imposed by the way morphisms are attached to objects and the way they compose. Such maps let you find similarities between categories and catalog different kinds of categories. That's when things get really interesting.

Functors

A functor, `F`, is a map from one category to another: it maps objects into objects and morphisms into morphisms. But it can't do it in a haphazard way because that would destroy the very structures that we are after. So we must impose some "obvious" (mathematicians love that word) constraints.

First of all, if you have a morphism between two objects in the first category then it better be mapped into a morphism between the corresponding objects in the second category. Fig 2 explains this diagrammatically. Object A is mapped into F(A), object B into F(B). A morphism f from A to B is mapped into a morphism F(f) from F(A) to F(B). Mathematicians say that such a diagram must commute, that is the result must be the same whether you go from A to F(A) and then apply F(f), or first apply f and then go from B to F(B).

$$\begin{tikzcd}
F(A) \arrow[r, "F(f)"] & F(B) \\
A \arrow[u] \arrow[r, "f"] & B \arrow[u]
\end{tikzcd}$$

Fig 2. Diagram showing the action of a functor F on objects A and B and a morphism f. The bottom part lives in F's domain (source) category, the top part in its codomain (the target).

Moreover, such mapping should preserve the composition property of morphisms. So if morphism `h` is a composition of `f` and `g`, then `F(h)` must be a composition of `F(f)` and `F(g)`. And, of course, the functor must map identity morphisms into identity morphisms.

To get a feel for how constrained functors are by these conditions, consider how you could map the category in Fig 1 into itself (such a functor just rearranges things inside one category). There are two trivial mappings that collapse both objects into one (either A or B), and turn all morphisms into identity. Then there is the identity functor that maps both objects into themselves and all morphisms into themselves. Finally, there is just one "interesting" functor that maps A into B and B into A with f and g switching roles. Now imagine a similar category but with the g arrow removed (yes, it's still a category). Suddenly there is no functor other than the collapsing ones between Fig 1 and that new category. That's because the two categories have completely different structure.

Let me now jump into more familiar territory. Since we are mostly interested in one category, **Hask**, let me define a functor that maps that category into itself (such functors are called endofunctors). An object in **Hask** is a type, so our functor must map types into types. The way to look at it is that a functor in **Hask** constructs one type from another– it's a *type constructor*. Don't get confused by the name: a type constructor creates a new type in your program, but that type has already existed in **Hask**.

A classical example is the *list* type constructor. Given any type it constructs a list of that type. Type `Integer` is mapped into list of integers or, in Haskell notation, `[Integer]`. Notice that this is not a map defined on integer *values*, like 1, 2, or 3. It also doesn't *add* a new type to **Hask**— the type `[Integer]` is already there. It just maps one type into another. For C++ programmers: think of mapping type T into a container of T; for instance, `std::vector<T>`.

Mapping the types is the easy part, what about functions? We have to find a way to take a particular function and map it into a function on lists. That's also easy: apply the function to each element of the list in turn. There is a (higher level) function in Haskel that does it. It's called `map` and it takes a function and a list and returns a new list (or, because of currying, you may say that it takes a function and returns a function acting on lists). In C++ there is a corresponding template function called `std::transform` (well, it takes two iterators and a function object, but the idea is the same).

Mathematicians often use diagrams to illustrate the properties of morphisms and functors (see Fig 2). The arrows for morphisms are usually horizontal, while the arrows for functors are vertical (going up). That's why the mapping of morphisms under a functor is often called *lifting*. You can take a function operating on integers and "lift it" (using a functor) to a function operating on lists of integers, and so on.

The list functor obviously preserves function composition and identity (I'll leave it as an easy but instructive exercise for the reader).

And now for another moment of Zen. What's the second most important property of programming? Reusability! Look what we have just done: We took all the functions we've implemented so far and lifted them to the level of lists. We've got functions operating on lists essentially for free (well, we've got a small but important subset of those functions). And the same trick may be applied to all kinds of containers, arrays, trees, queues, `unique_ptr`s and more.

It's all beautiful, but you don't really need category theory to apply functions to lists. Still it's always good to see patterns in programming, and this one is definitely a keeper. The real revolution starts with monads. And, guess what, the list functor is actually a monad. You just need a few more ingredients.

What's the intuition behind the statement that mappings expose the structure of the system? Consider the schematic of the London underground in Fig 3. It's just a bunch of circles and lines. It's only relevant because there is a mapping between the city of London and this schematic. The circles correspond to tube stations and the lines to train connections. Most importantly, if trains run between two stations, the corresponding circles in the diagram are connected by lines and vice versa: these are the constraints that the mapping preserves. The schematic shows a certain structure that exists in London (mostly hidden underground) which is made apparent by the mapping.

Fig 3. The schematic map of London underground system.

Interestingly, what I'm doing here is also mapping: London and the underground map correspond to two categories. Trains stations/circles are objects and train connections/lines are morphism. How's that for an example?

Endofunctors

Mathematicians love mappings that preserve "obvious" constraints. As I explained, such mappings abstract inner structures away from the details of implementation. But you can also learn a lot about structure by studying non-trivial mappings into itself. Functors that map a category into itself are called endofunctors (like *endo*-scopes they let you look inside things). If functors expose similarities, endofunctors expose self-similarities. Take one look at the fractal fern, Fig 4, and you'll understand how powerful self-similarity can be.

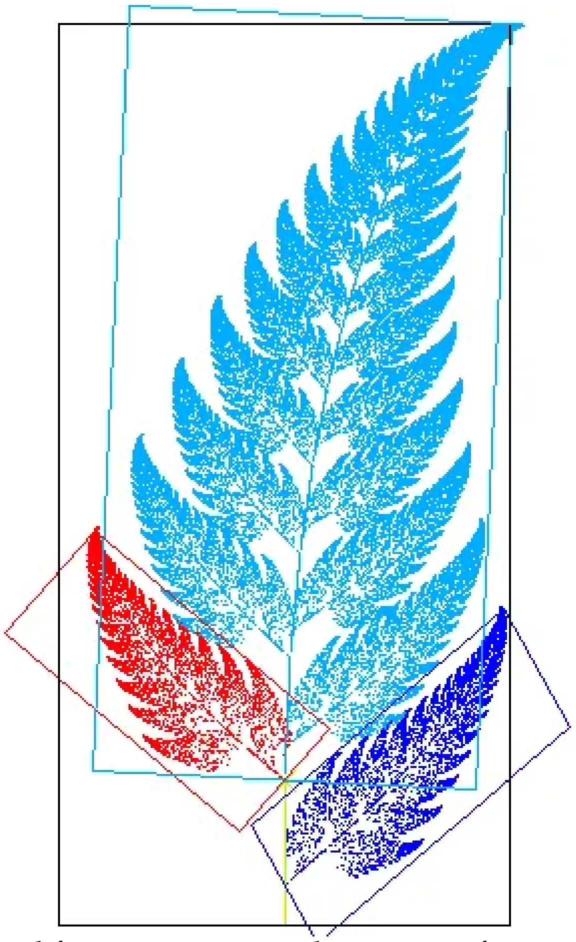
Fig 4. This fractal fern was generated using just four endomorphisms.

With a little bit of imagination you can see the list functor exposing fern-like structures inside **Hask** (Fig 5). Chars fan out into lists of Chars, which then fan out into lists of lists of Chars, and so on, ad infinitum. Horizontal structures described by functions from `Char` to `Bool` are reflected at higher and higher levels as functions on lists, lists of lists, etc.

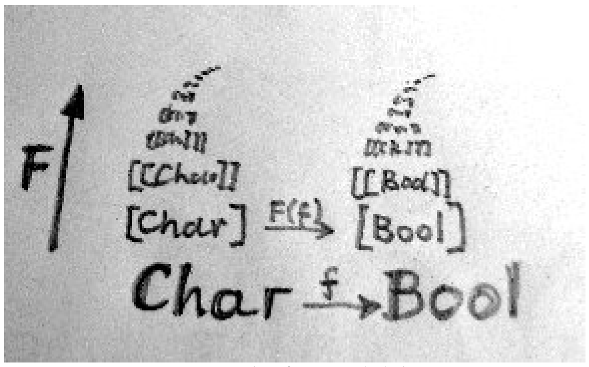

Fig 5. The action of the list type constructor reveals fractal-like structure inside Hask. The functor lifts things up, the functions act horizontally.

A C++ template that takes a type parameter could be considered a type constructor. How likely is it that it also defines a functor (loosely speaking– C++ is not as mathematized as Haskell)? You have to ask yourself: Is the type parameter constrained in any way? It's often hard to say, because type constraints are implicit in the body of a template and are tested only during instantiation. For instance, the type parameter for a `std::vector` must be copyable. That eliminates, for instance, classes that have private or `deleted` (in C++0x) copy constructors. This is not a problem though, because copyable types form a subcategory (I'm speaking really loosely now). The important thing is that a vector of copyable is itself copyable, so the "endo-" part of the endomorphism holds. In general you want to be able to feed the type created by the type constructor back to the type constructor, as in `std::vector<std::vector<Foo>>`. And, of course, you have to be able to lift functions in a generic way too, as in `std::transform`.

Monads

Ooh, Monads!
–Haskell Simpson

It's time to finally lift the veil. I'll start with the definition of a monad that builds on the previous sections and is mostly used by mathematicians. There is another one that's less intuitive but easier to use in programming. I'll leave that one for later.

A monad is an endofunctor together with two special families of morphisms, both going vertically, one up and one down (for "directions" see Fig 5). The one going up is called *unit* and the one going down is called *join*.

Now we are juggling a lot of mappings so let's slow down to build some intuition. Remember, a functor maps objects: in our case, types, which are sets of values. The functor doesn't see what's inside the objects; morphisms, in general, do. In our case, a morphism is a function that maps values of one type into values of another type. Our functors, which are defined by type constructors, usually map poorer types into richer types; in the sense that type Bool is a set that contains just two elements, True and False, but type [Bool] contains infinitely many lists of True and False.

Unit takes a value from the poorer type, then picks one value from the richer type, and pronounces the two roughly equivalent. Such a rough equivalent of True from the Bool object is the one-element list [True] from the [Bool] object. Similarly, unit would map False into [False]. It would also map integer 5 into [5] and so on.

Unit can be thought of as immersing values from a lower level into the higher level in the most natural way possible. By the way, in programming we call a family of functions defined for any type a polymorphic function. In C++, we would express unit as a template, like this:

```
template<class T>
std::vector<T> unit(T value) {
    std::vector<T> vec;
    vec.push_back(value);
    return vec;
}
```

To explain *join*, imagine the functor acting twice. For instance, from a given type `T` the list functor will first construct the type `[T]` (list of `T`), and then `[[T]]` (list of list of `T`). Join removes one layer of "listiness" by joining the sub-lists. Plainly speaking, it just concatenates the inner lists. Given, for instance, `[[a, b], [c], [d, e]]`, it produces `[a, b, c, d, e]`. It's a many-to-one mapping from the richer type to the poorer type and the type-parameterized family of joins also forms a polymorphic function (a template, in C++).

There are a few monadic axioms that define the properties of unit and join (for instance that unit and join cancel each other), but I'm not going to elaborate on them. The important part is that the existence of unit and join imposes new constraints on the endofunctor and thus exposes even more structure.

Mathematicians look at `join` as the grandfather of all multiplication with `unit` being its neutral element. It's heaven for mathematicians because multiplication leads to algebraic structures and indeed monads are great for constructing algebras and finding their hidden properties.

Unlike mathematicians, we programmers are not that interested in algebraic structures. So there must be something else that makes monads such a hit. As I mentioned in the beginning, in programming we often face problems that don't naturally translate into the functional paradigm. There are some types of computations that are best expressed in imperative style. It doesn't mean they can't be translated into functions, it's just that the translation is somewhat awkward and tedious. Monads provide an elegant tool to do this translation. Monads made possible the absorption and assimilation of imperative programming into functional programming, so much so that some people claim (tongue in cheek?) that Haskell is the best imperative language. And like all things functional monads are bound to turn around and find their place in imperative programming. But that's material for my next blog post.

Bibliography

- The Catsters. An excellent series of videos about category theory.
- Mike Vanier, Yet Another Monad Tutorial. Explains Haskel monads in great detail.
- Dan Pipone, Neighborhood of Infinity. Interesting blog with a lot of insights into category theory and Haskell.
- Eugenio Moggi, Notions of Computation and Monads. This is a hard core research paper that started the whole monad movement in functional languages.
- Philip Wadler, Monads for Functional Programming. The classic paper introducing monads into Haskell.

41 Responses to "Monads for the Curious Programmer, Part 1"

1. Brendan Miller Says:

   January 25, 2011 at 2:28 pm
   Great post! I'm curious to learn more about haskell and monads.

2. Quentin Moser Says:

   January 25, 2011 at 5:19 pm
   It's rare and nice to see an introduction to monads that takes its time, rather than jumping out of nowhere to the author's personal epiphany. This leaves me eager to read the next part!

3. Saji Says:

   January 25, 2011 at 6:28 pm
   An example demoing the clean partitioning of stateful vs stateless parts of a simple Haskell program would be helpful. I know monads are more general than that but I think it would be very helpful for many like me.

4. Jason Kostempski Says:

   January 25, 2011 at 9:52 pm
   Monads never really baffled me because my first exposure to them was through Brian Beckman http://channel9.msdn.com/Tags/brian+beckman . Watch them all, monad related or not. This man has an amazing way of transferring his personal understanding of things directly into your brain. He's directly involved in the making of LINQ and BizTalk. Both of which are based heavily on Category Theory.

5. mrkkrj Says:

   January 26, 2011 at 1:49 am
   >If monads are so important in functional programming, they must also pop up in C++ metaprogramming. And indeed they do.

   well, that sounds ineotersting! Can't wait to read about it.

6. Ábel Sinkovics Says:

   January 26, 2011 at 7:17 am
   I've been following your blog for a while and I completely agree with the message about the connection of functional programming and C++ template metaprograms.

   At Eötvös Loránd University, Budapest there is a research group working with template metaprograms since 2003. The group has implemented template metaprogram debugger and visualization tool and a few other applications. The long term goal is writing metaprograms in Haskell, transforming them into C++ in automated way and keep C++ syntax only as a kind of assembly of metaprograms.

   As a part of this effort, I've written a template metaprogramming parser generator library (http://abel.sinkovics.hu/?mod=kutat) which can parse a text at C++ compilation time. It can be used to implement domain specific languages in C++ without external tools. You can do really cool things using it, but error messages provided by the parsers built with my library are not really usable.

   Due to the connection between functional programming and C++ template metaprogramming, I've been trying to improve the error messages using monads. I've been playing with monads in C++ template metaprograms and I'm building something similar to Haskell's do-syntax for template metaprogramming. The plan is to use that syntax for parser construction. I already have some parts working, with my current implementation you can write things like:

```
struct x;

template <.....>
struct my_metafunction_using_do_syntax :
  DO<
    SET<x, .....>,
    CALL<some_other_metafunction, x>
  >
{};
```

You can find my code here: https://github.com/sabel83/mpllibs
I recommend looking at the unit tests for my do-syntax implementation to see it in action: src/lib/mpllibs/error/test/do_.cpp

Note that the monadic part is not documented yet, since it is still work in progress. I'm curious about other uses of monads in C++ template metaprogramming.

7. Ábel Sinkovics Says:

   January 26, 2011 at 7:25 am
   The code example in my previous comment doesn't seem to appear well. You can find it here: http://sinkovics.hu/sample_cpp.html

8. Bartosz Milewski Says:

   January 26, 2011 at 11:27 am
   @Ábel: Thanks for the link. Interesting paper about DLS parser generators. You never mention monads explicitly, but you define return_ and the way to combine parsers that's a direct translation of >>=. Eric Niebler, the author and Proto, and I are good friends, so I have keen interest in DSLs and parsers. I'll elaborate on this in the third installment of my tutorial. [I reformatted the code in your post. The trick is to use <pre> and replace angle brackets with &lt; and &gt;.]

9. Bartosz Milewski Says:

   January 26, 2011 at 11:30 am
   @Jason: Yes, I've seen Brian's videos. I like his emphasis on composability.

10. moo Says:

    January 26, 2011 at 12:03 pm
    Very much appreciated

11. » links for 2011-01-26 (Dhananjay Nene) Says:

    January 26, 2011 at 1:01 pm
    […] Monads for the Curious Programmer, Part 1 « Bartosz Milewski's Programming Cafe Some catchy statements here. Monads for the Curious Programmer, Part 1 http://ff.im/-wPCDO (tags: via:packrati.us) […]

12. Paul Johnson Says:

January 26, 2011 at 8:49 pm
Looking for the join which maps [Article on Monads] to(roughly) Article on Monads.

13. Monads for the Curious Programmer, Part 1 Says:

    January 26, 2011 at 8:52 pm
    […] –Monad Te Ching I don't know if I'm exaggerating but it seems like every… [full post] Bartosz Milewski Bartosz Milewski's Programming Cafe c++category theoryfunctional […]

14. Andres Kievsky Says:

    February 7, 2011 at 9:53 pm
    Great post – when can we expect part 2? Thanks!

15. Monads for the Curious Programmer: Part 2 « Bartosz Milewski's Programming Cafe Says:

    March 14, 2011 at 9:37 am
    […] my previous post I talked about categories, functors, endofunctors, and a little about monads. Why didn't I […]

16. Brett Says:

    March 15, 2011 at 6:15 pm
    A little late to the game, I know…

    First, forgive my ignorance, here… category theory in unfamiliar to me, but I find the whole topic rather fascinating, and I thought I was following along pretty well.

    And then I got to your definition of list as a monad.

    You mention the functor F which maps T -> [T], and f(t) -> map(f(t)). This makes perfect sense to me.

    Now, a monad for the list would be defined as F (mapping T -> [T]), a unit morphism f(t) = [t], and then some sort of join function. Your definition defines join([t]) = concat([t]).

    But I'm puzzled. Your definition of join certainly maps [[t]] -> [t], which meets your many-to-one definition. But there's no definition, here, that maps [t] -> t. So while you have a morphism to lift T -> [T] for all T, there is no corresponding join function for all T.

    So I feel like I must be missing something here…

    As an aside: *great article*. This is fascinating stuff!

17. Bartosz Milewski Says:

    March 15, 2011 at 6:36 pm
    Join accepts lists of lists. You can't call join on just any list. When I said that unit and join cancel each other I meant that when you act with unit on a list, you get a list of lists. If you join that list of lists, you get back your original list. For instance, unit [a, b, c] = [[a, b, c]]. join [[a, b, c]] = [a, b, c].

18. Brett Says:

    March 15, 2011 at 7:07 pm
    Yup, that much makes sense… I think what had me a little puzzled is that, in the description, unit and join are depicted as as opposites, “one going up, one going down”. But as you say, that’s not 100% true. In the example of lists, unit maps t -> [t] for all t in T. But join obviously can’t do the reverse for all [t], but only for [t] where t is in [u].

    And the more I think about it, the more that makes sense. Take the type IO a: one of the first things you discover about Haskell is that you can’t go from IO a -> a. But knowing this limitation of join, that’s actually not at all surprising.

    Anyway, no big deal, I just felt like I *had* to be missing something… fortunately for me, it looks like I’m not! 🙂

    BTW, thanks for the quick response, and apologies if this is all painfully obvious to everyone else. 🙂

19. csoroz Says:

    March 16, 2011 at 6:02 am
    Minor typo:

    Functor
    …
    A morphism f from A to B is mapped into a morphism F(f) from F(A) to F(b).

    Sould be F(B) instead of F(b).

20. Bartosz Milewski Says:

    March 16, 2011 at 10:27 am
    Good catch, thanks!

21. Jeffrey Griffin Says:

    March 23, 2011 at 7:33 pm
    Excellent writing and presentation!

    As a side note, I feel that it would have been a little more clear if the box containing Fig. 3 were placed right after the paragraph just before the start of the “Functors” section. I was able to figure out what was going on, but everything else flowed so perfectly I thought I might as well mention it.

    Also, on the third paragraph in the ‘Monads’ section, where it says,

    “Our functors, which are defined by type constructors, usually map poorer types into richer types; in the sense that Bool contains just two elements, True and False, but [Bool] contains infinitely many lists.”

    Am I misreading that, or should it say that [Bool] contains infinitely many elements (or something of that sort)?

22. Bartosz Milewski Says:

    March 24, 2011 at 9:17 am
    @Jeffrey: Good points. I moved the box and changed the wording. I was talking about types as sets of values. Bool has only two values, but [Bool] is a set of infinitely many values, which are all possible lists of True and False.

23. Burak Ekici Says:

    April 12, 2011 at 8:31 am
    Dear Dr. Milewski,

    I want to prove Haskell's Cont data constructor + 2 natural transformations (bind + return) construct a monad via showing that it satisfies following properties :

    join . fmap join = join . join
    join . fmap return = join . return = id
    return . f = fmap f . return
    join . fmap (fmap f) = fmap f . join

    Because of that reason, I need definition of Cont monad with fmap, join and return.
    However,I was not able to define it.

    Could you please help me, if you have time?

    Bests,
    Burak Ekici.

24. Bartosz Milewski Says:

    April 12, 2011 at 9:14 am
    @Burak: I don't know it off the top of my head, but I'm sure you can find it on the internet if you search for "continuation monad".

25. Sudoku Solver in Haskell » #AltDevBlogADay Says:

    May 23, 2011 at 6:05 am
    […] for the Curious Programmer: https://bartoszmilewski.wordpress.com/2011/01/09/monads-for-the-curious-programmer-part-1/ https://bartoszmilewski.wordpress.com/2011/03/14/monads-for-the-curious-programmer-part-2/ […]

26. Monads in C++ « Bartosz Milewski's Programming Cafe Says:

    July 11, 2011 at 1:12 pm
    […] Part 1 […]

27. Plouj Says:

    July 24, 2011 at 2:48 pm
    Minor typo: "unit an join cancel each other" – replace "an" with "and"

28. Bartosz Milewski Says:

    July 25, 2011 at 9:48 am
    @Plouj: Done!

29. Issue 34 – The Crack of Real-Time Information — TLN Says:

    January 13, 2013 at 9:36 pm
    […] Monads for the Curious Programmer I have shared links to tutorials on monads in the past but this is the first one I read that actually really made sense to me. Using mathematics and functions the author does a superb job of breaking monads down in a clear and easy to understand way. If you do any functional programming this is definitely worth a read. […]

30. Zafer Sernikli Says:

    April 30, 2013 at 2:00 pm
    Excellent post! But, just to clarify, I have to mention that a function must defined to real numbers. Mathematically talking, if the image set of a function (not a function indeed) is not real numbers, we call it a "map", not a function. Functions are a subset of maps which have image sets as real numbers, i.e. a map is function iff its image set is real numbers.

31. Bartosz Milewski Says:

    May 1, 2013 at 3:12 pm
    @Zafer : I think function is commonly used in mathematics as a synonym for map. What you're describing is a real functions. There are also complex functions, etc. The definition I'm familiar with is that a function is relation with the property that for each element of the domain there is at most one element in the image that satisfied that relation. In other words, a function is a single-valued relation between two sets.

32. Zafer Sernikli Says:

    January 29, 2014 at 12:48 pm
    @Bartosz : Sorry for the latency. The usage you told is a wrong usage. We sometimes call maps as functions but some of them are not. For instance, a complex function is a map from the set of complex numbers to real numbers, not a map from complex numbers to complex numbers. "A single-valued relation between two sets" is a map, and if the range set is the real numbers, then it is also a function.

    It was only a clarification, actually most of the matematicians uses the word "function" wrongly as well.

33. Bartosz Milewski Says:

    January 29, 2014 at 1:00 pm
    @Zafer: I guess I'm with the majority of mathematicians who use the word "function" wrongly 🙂

34. Daniel Sparks Says:

    May 25, 2014 at 2:57 pm

@Bartosz, As a mathematician taking a transition into programming, I found this article just right. It conveyed exactly what I wanted to know about monads in a very smooth read.

@Zafer, I hope you're not insinuating that a function must map into the real numbers. This may be a convention in some analysis or manifolds textbooks, but is not the general terminology.

To be absolutely clear, a function can map from any set to any set. A real-valued function can map any set into the reals. A map can mean anything you like, for instance: a morphism in a category, a set-theoretic function, or even something very "functional" like a map from a manifold into its base field or a linear functional on a vector space. And remember, the majority of the time, morphisms in categories are literally functions + other structure, but sometimes they are not. So either way, to say that a function is not a map or that a map is not a function is not right.

35. Chang, Jaehyeok Says:

    October 5, 2015 at 10:27 am
    This is best article for the monad, I've ever seen.
    [[[[[[[Thanks]]]]]]] a lot !!

    someone from Korea

36. R.Bullington-McGuire (@obscurerichard) Says:

    March 12, 2016 at 5:53 am
    Thank you for writing this. Here's one small correction:

    Unit can be though of => Unit can be thought of

37. Marcel Moosbrugger Says:

    June 3, 2016 at 12:34 pm
    Very nice explanation. The abstract view from mathematics onto programming makes a lot of things easier to understand in general. Some time ago I wrote a blog about monads in haskell and their link to the imperative world, but was way more specific then you are. I'd really like to get a feedback from you.

38. kayvan Says:

    October 8, 2016 at 2:49 pm
    Thank you for your explanation as it really helped me get a better intuition of Monads. Watching your Bifunctor video was a good "prep" for this article.

39. Chris Schroeder Says:

    June 28, 2017 at 11:54 pm
    cool write-up and good comments too. thanks!

40. Cloudsdale Says:

    May 23, 2018 at 7:16 pm
    > *"You can easily check that any two morphisms can be composed"*

Hmm so what is `i_A` composed with `i_B` then? :q

41. Bartosz Milewski Says:

May 23, 2018 at 11:53 pm

You got me there! 😉

This site uses Akismet to reduce spam. Learn how your comment data is processed.