How to extract value from monadic action

Is there a built-in function with signature `:: (Monad m) => m a -> a` ?

Hoogle tells that there is no such function.

Can you explain why?

haskell monads comonad

edited Oct 18 '14 at 14:01
Zoidberg
17.5k ●4 ●34 ●69

asked Dec 19 '11 at 21:22
ДМИТРИЙ МАЛИКОВ
10.8k ●5 ●35 ●89

2 Related: taking out a value out of a monad? haskell – Jan Dec 19 '11 at 21:26

13 There isn't, but there is a function that turns functions expecting an `a` into functions expecting an `m a`: `(=<<) :: Monad m => (a -> m b) -> (m a -> m b)`. Invert your expectations, and you will be fine. =) – Daniel Wagner Dec 19 '11 at 21:33

2 In the same vein as what Daniel Wagner said, `liftM :: Monad m => (a -> b) -> (m a -> m b)` allows a "regular" function to accept a monadic value as input, but in exchange it must output a monadic value rather than a "regular" value. – Dan Burton Dec 19 '11 at 22:39

7 Answers

A monad only supplies two functions:

```
return :: Monad m => a -> m a
(>>=) :: Monad m => m a -> (a -> m b) -> m b
```

Both of these return something of type `m a`, so there is no way to combine these in any way to get a function of type `Monad m => m a -> a`. To do that, you'll need more than these two functions, so you need to know more about `m` than that it's a monad.

For example, the `Identity` monad has `runIdentity :: Identity a -> a`, and several monads have similar functions, but there is no way to provide it generically. In fact, the inability to "escape" from the monad is essential for monads like `IO`.

answered Dec 19 '11 at 21:31
hammar
107k ●8 ●219 ●325

There is probably a better answer than this, but one way to see why you cannot have a type `(Monad m) => m a -> a` is to consider a null monad:

```
data Null a = Null

instance Monad Null where
    return a = Null
    ma >>= f = Null
```

Now `(Monad m) => m a -> a` means `Null a -> a`, ie getting something out of nothing. You can't do that.

answered Dec 19 '11 at 21:29

Owen
19.3k ● 4 ● 51 ● 89

4 On the other hand this fact doesn't prevent `fromJust` from existence. It returns contents of a `Just` and raises an exception in case of `Nothing`. Your `Null` monad could simply always raise an exception on calling such imaginary monad unwrapping function. – Jan Dec 19 '11 at 21:32

11 @Jan: On the other other hand that doesn't make either function a good idea. `fromJust` is terrible and would be better not existing. – C. A. McCann Dec 19 '11 at 21:51

1 @C.A.McCann I +1 your comment partly because `fromJust` is bad... but mainly because you used "other other hand". – Adam Wagner Dec 19 '11 at 23:54

Agee with C.A. McCann; the fact that the proposed method for the `Monad` class would have to be implemented as raising an exception for many classes is evidence enough that it should not be part of the class. If we actually had some examples of "monad that we can extract from" that we wanted to cope, then we could create a subclass of Monad to tackle them and put the operation there. – Luis Casillas Dec 20 '11 at 0:26

`fromJust` has its uses, just not in robust code. I use it when there's no point in continuing on failure. For example, when the UI definition file is missing for an mplayer frontend. – György Andrasek Dec 20 '11 at 3:59

This doesn't exist because `Monad` is a pattern for composition, not a pattern for decomposition. You can always put more pieces together with the interface it defines. It doesn't say a thing about taking anything apart.

Asking why you can't take something out is like asking why Java's `Iterator` interface doesn't contain a method for adding elements to what it's iterating over. It's just not what the `Iterator` interface is for.

And your arguments about specific types having a kind of extract function follows in the exact same way. Some particular implementation of `Iterator` might have an `add` function. But since it's not what `Iterator`s are for, the presence that method on some particular instance is irrelevant.

And the presence of `fromJust` is just as irrelevant. It's not part of the behavior `Monad` is intended to describe. Others have given lots of examples of types where there is no value for `extract` to work on. But those types still support the intended semantics of `Monad`. This is important. It means that `Monad` is a more general interface than you are giving it credit for.

answered Dec 19 '11 at 21:56

Carl
15.2k ● 2 ● 38 ● 57

> Is there a build-in function with signature `:: (Monad m) => m a -> a` ?

If Hoogle says there isn't...then there probably isn't, assuming your definition of "built in" is "in the base libraries".

> Hoogle tells that there is no such function. Can you explain why?

That's easy, because Hoogle didn't find any function in the base libraries that matches that type signature!

More seriously, I suppose you were asking for the monadic explanation. The issues are *safety* and *meaning*. (See also my previous thoughts on `magicMonadUnwrap :: Monad m => m a -> a`)

Suppose I tell you I have a value which has the type `[Int]`. Since we know that `[]` is a monad, this is similar to telling you I have a value which has the type `Monad m => m Int`. So let's suppose you want to get the `Int` out of that `[Int]`. Well, which `Int` do you want? The first one? The last one? What if the value I told you about is actually an empty list? In that case, there isn't even an `Int` to give you! So for lists, it is *unsafe* to try and extract a single value willy-nilly like that. Even when it is safe (a non-empty list), you need a list-specific function (for example, `head`) to clarify what you *mean* by desiring `f :: [Int] -> Int`. Hopefully you can intuit from here that the *meaning* of `Monad m => m a -> a` is simply not well defined. It could hold multiple meanings for the same monad, or it could mean absolutely nothing at all for some monads, and sometimes, it's just simply not safe.

answered Dec 19 '11 at 23:02

Dan Burton
31.3k ● 13 ● 85 ● 159

1 I don't see how `Monad m => m a -> a` lacks meaning in any way that wouldn't also apply to `>>=` or `return` or `fail`. You could always say that the "meaning" of the operation isn't well-defined in advance of knowing the full implementation that `m` provides for its inclusion in the Monad type class. For your list example, a function with type `Monad m => m a -> a` could very well mean any of the things you suggest -- and *any* of them *might* be valid. You could always use `newtype` to tweak the behavior for *your* application, but denying even the chance to do it seems too severe. – Mr. F Dec 21 '14 at 0:14

Suppose there was such a function:

```
extract :: Monad m => m a -> a
```

Now you could write a "function" like this:

```
appendLine :: String -> String
appendLine str = str ++ extract getLine
```

Unless the `extract` function was guaranteed never to terminate, this would violate referential transparency, because the result of `appendLine "foo"` would (a) depend on something other than `"foo"`, (b) evaluate to different values when evaluated in different contexts.

Or in simpler words, if there was an actually useful `extract` operation Haskell would not be purely functional.

answered Dec 20 '11 at 0:19

Luis Casillas
19.1k ●2 ●29 ●75

1 unsafePerformIO does just this. – Mr. F Dec 21 '14 at 0:28

1 Don't use that! Boooooooo!!!!!! – Luis Casillas Dec 22 '14 at 21:33

Because it may make no sense (actually, *does* make no sense in many instances).

For example, I might define a Parser Monad like this:

```
data Parser a = Parser (String ->[(a, String)])
```

Now there is absolutely no sensible default way to get a `String` out of a `Parser String`. Actually, there is no way at all to get a String out of this with just the Monad.

answered Dec 19 '11 at 21:27

Cubic
7,231 ●2 ●18 ●52

Well, technicaly there is unsafePerformIO for the IO monad.

But, as the name itself suggests, this function is evil and you should only use it if you *really* know what you are doing (and if you have to ask wether you know or not then you don't)

answered Dec 20 '11 at 20:53

hugomg
38.2k ●9 ●71 ●148

6 You shouldn't tell innocent people how to do evil stuff! – is7s Dec 20 '11 at 21:17

1 There is also `unsafeHead` for the List monad...(oh wait, it's just called `head`...but it is similarly, though not quite as drastically, unsafe.) – Dan Burton Dec 21 '11 at 5:23

-1 It doesn't answer the OP question. – mb14 Oct 18 '14 at 16:18