What does it mean for a monad to be strong?

Dan Piponi

June 2024 (originally August 2023)

This is something I put on github years ago but I probably should have put it here.

Here's an elementary example of the use of the list monad:

```
test1 = do
  x <- [1, 2]
  y <- [x, 10*x]
  [x*y]
```

We can desugar this to:

```
test2 = [1, 2] >>= \x -> [x, 10*x] >>= \y -> [x*y]
```

It looks like we start with a list and then apply a sequence (of length 2) of functions to it using bind `(>>=)`. This is probably why some people call monads workflows and why the comparison has been made with Unix pipes.

But looks can be deceptive. The operator `(>>=)` is right associative and `test2` is the same as `test3`:

```
test3 = [1, 2] >>= (\x -> [x, 10*x] >>= \y -> [x*y])
```

You can try to parenthesise the other way:

```
-- test4 = ([1, 2] >>= \x -> [x, 10*x]) >>= \y -> [x*y]
```

We get a "Variable not in scope: x" error. So `test1` doesn't directly fit the workflow model. When people give examples of how workflow style things can be seen as monads they sometimes use examples where later functions don't refer to variables defined earlier. For example at the link I gave above the line `m >>= \x -> (n >>= \y -> o)` is transformed to `(m >>= \x -> n) >>= \y -> o` which only works if `o` makes no mention of `x`. I found similar things to be true in a number of tutorials, especially the ones that emphasise the Kleisli category view of things.

But we can always "reassociate" to the left with a little bit of extra work. The catch is that the function above defined by `\y -> ...` "captures" `x` from its environment. So it's not just one function, it's a family of functions parameterised by `x`. We can fix this by making the dependence on `x` explicit. We can then pull the inner function out as it's no longer implicitly dependent on its immediate context. When compilers do this it's called lambda lifting.

Define (the weirdly named function) `strength` by

```
strength :: Monad m => (x, m y) -> m (x, y)
strength (x, my) = do
  y <- my
  return (x, y)
```

It allows us to smuggle `x` "into the monad".

And now we can rewrite `test1`, parenthesising to the left:

```
test5 = ([1, 2] >>= \x -> strength (x, [x, 10*x]))
                >>= \(x, y) -> [x*y]
```

This is much more like a workflow. Using `strength` we can rewrite any (monadic) `do` expression as a left-to-right workflow, with the cost of having to throw in some applications of `strength` to carry along all of the captured variables. It's also using a composition of arrows in the Kleisli category.

A monad with a strength function is called a strong monad. Clearly all Haskell monads are strong as I wrote `strength` to work with any Haskell monad. But not all monads in category theory are strong. It's a sort of hidden feature of Haskell (and the category Set) that we tend not to

refer to explicitly. It could be said that we're implicitly using strength whenever we refer to earlier variables in our `do` expressions.

See also nlab.

```
main = do
  print test1
  print test2
  print test3
  -- print test4
  print test5
```