Functors, Applicatives, and Monads

PUBLISHED JANUARY 3, 2017

This content originally appeared on School of Haskell. Thanks for Julie Moronuki for encouraging me to update/republish, and for all of the edits/improvements.

NOTE Code snippets below can be run using the Stack build tool, by saving to a file `Main.hs` and running with `stack Main.hs`. More information is available in the How to Script with Stack tutorial.

Let's start off with a very simple problem. We want to let a user input his/her birth year, and tell him/her his/her age in the year 2020. Using the function `read`, this is really simple:

```
#!/usr/bin/env stack
-- stack --resolver lts-7.14 --install-ghc runghc
main = do
    putStrLn "Please enter your birth year"
    year <- getLine
    putStrLn $ "In 2020, you will be: " ++ show (2020 - read year)
```

If you run that program and type in a valid year, you'll get the right result. However, what happens when you enter something invalid?

```
Please enter your birth year
hello
main.hs: Prelude.read: no parse
```

The problem is that the user input is coming in as a `String`, and `read` is trying to parse it into an `Integer`. But not all `String`s are valid `Integer`s. `read` is what we call a **partial function**, meaning that under some circumstances it will return an error instead of a valid result.

A more resilient way to write our code is to use the `readMaybe` function, which will return a `Maybe Integer` value. This makes it clear with the types themselves that the parse may succeed or fail. To test this out, try running the following code:

```
#!/usr/bin/env stack
-- stack --resolver lts-7.14 --install-ghc runghc
import Text.Read (readMaybe)

main = do
    -- We use explicit types to tell the compiler how to try and parse the
    -- string.
    print (readMaybe "1980" :: Maybe Integer)
    print (readMaybe "hello" :: Maybe Integer)
    print (readMaybe "2000" :: Maybe Integer)
    print (readMaybe "two-thousand" :: Maybe Integer)
```

So how can we use this to solve our original problem? We need to now determine if the result of `readMaybe` was successful (as `Just`) or failed (a `Nothing`). One way to do this is with pattern matching:

```
#!/usr/bin/env stack
-- stack --resolver lts-7.14 --install-ghc runghc
import Text.Read (readMaybe)

main = do
    putStrLn "Please enter your birth year"
    yearString <- getLine
    case readMaybe yearString of
        Nothing -> putStrLn "You provided an invalid year"
        Just year -> putStrLn $ "In 2020, you will be: " ++ show (2020 - year)
```

Decoupling code

This code is a bit coupled; let's split it up to have a separate function for displaying the output to the user and another separate function for calculating the age.

```
#!/usr/bin/env stack
-- stack --resolver lts-7.14 --install-ghc runghc
import Text.Read (readMaybe)

displayAge maybeAge =
    case maybeAge of
        Nothing -> putStrLn "You provided an invalid year"
        Just age -> putStrLn $ "In 2020, you will be: " ++ show age

yearToAge year = 2020 - year

main = do
    putStrLn "Please enter your birth year"
    yearString <- getLine
    let maybeAge =
            case readMaybe yearString of
                Nothing -> Nothing
                Just year -> Just (yearToAge year)
    displayAge maybeAge
```

This code does exactly the same thing as our previous version. But the definition of `maybeAge` in `main` looks pretty repetitive to me. We check if the parse year is `Nothing`. If it's `Nothing`, we return `Nothing`. If it's `Just`, we return `Just`, after applying the function `yearToAge`. That seems like a lot of line noise to do something simple. All we want is to conditionally apply `yearToAge`.

Functors

Fortunately, we have a helper function to do just that. `fmap`, or **functor mapping**, will apply some function over the value contained by a **functor**. `Maybe` is one example of a functor; another common one is a list. In the case of `Maybe`, `fmap` does precisely what we described above. So we can replace our code with:

```
#!/usr/bin/env stack
-- stack --resolver lts-7.14 --install-ghc runghc
import Text.Read (readMaybe)

displayAge maybeAge =
    case maybeAge of
        Nothing -> putStrLn "You provided an invalid year"
        Just age -> putStrLn $ "In 2020, you will be: " ++ show age

yearToAge year = 2020 - year

main = do
    putStrLn "Please enter your birth year"
    yearString <- getLine
    let maybeAge = fmap yearToAge (readMaybe yearString)
    displayAge maybeAge
```

Our code definitely got shorter, and hopefully a bit clearer as well. Now it's obvious that all we're doing is applying the `yearToAge` function over the contents of the `Maybe` value.

So what *is* a functor? It's some kind of container of values. In `Maybe`, our container holds zero or one values. With lists, we have a container for zero or more values. Some containers are even more exotic; the `IO` functor is actually providing an action to perform in order to retrieve a value. The only thing functors share is that they provide some `fmap` function which lets you modify their contents.

do-notation

We have another option as well: we can use `do`-notation. This is the same way we've been writing `main` so far. That's because- as we mentioned in the previous paragraph- `IO` is a functor as well. Let's see how we can change our code to not use `fmap`:

```
#!/usr/bin/env stack
-- stack --resolver lts-7.14 --install-ghc runghc
import Text.Read (readMaybe)

displayAge maybeAge =
    case maybeAge of
        Nothing -> putStrLn "You provided an invalid year"
        Just age -> putStrLn $ "In 2020, you will be: " ++ show age

yearToAge year = 2020 - year

main = do
    putStrLn "Please enter your birth year"
    yearString <- getLine
    let maybeAge = do
            yearInteger <- readMaybe yearString
            return $ yearToAge yearInteger
    displayAge maybeAge
```

Inside the `do`-block, we have the **slurp operator** `<-`. This operator is special for `do`-notation and is used to pull a value out of its wrapper (in this case, `Maybe`). Once we've extracted the value, we can manipulate it with normal functions, like `yearToAge`. When we complete our `do`-block, we have to return a value wrapped up in that container again. That's what the `return` function does.

`do`-notation isn't available for all `Functor`s; it's a special feature reserved only for `Monad`s. `Monad`s are an extension of `Functor`s that provide a little extra power. We're not really taking advantage of any of that extra power here; we'll need to make our program more complicated to demonstrate it.

Dealing with two variables

It's kind of limiting that we have a hard-coded year to compare against. Let's fix that by allowing the user to specify the "future year." We'll start off with a simple implementation using pattern matching and then move back to do-notation.

```
#!/usr/bin/env stack
-- stack --resolver lts-7.14 --install-ghc runghc
import Text.Read (readMaybe)

displayAge maybeAge =
    case maybeAge of
        Nothing -> putStrLn "You provided invalid input"
        Just age -> putStrLn $ "In that year, you will be: " ++ show age

main = do
    putStrLn "Please enter your birth year"
    birthYearString <- getLine
    putStrLn "Please enter some year in the future"
    futureYearString <- getLine
    let maybeAge =
            case readMaybe birthYearString of
                Nothing -> Nothing
                Just birthYear ->
                    case readMaybe futureYearString of
                        Nothing -> Nothing
                        Just futureYear -> Just (futureYear - birthYear)
    displayAge maybeAge
```

OK, it gets the job done... but it's very tedious. Fortunately, do-notation makes this kind of code really simple:

```
#!/usr/bin/env stack
-- stack --resolver lts-7.14 --install-ghc runghc
import Text.Read (readMaybe)

displayAge maybeAge =
    case maybeAge of
        Nothing -> putStrLn "You provided invalid input"
        Just age -> putStrLn $ "In that year, you will be: " ++ show age

yearDiff futureYear birthYear = futureYear - birthYear

main = do
    putStrLn "Please enter your birth year"
    birthYearString <- getLine
    putStrLn "Please enter some year in the future"
    futureYearString <- getLine
    let maybeAge = do
            birthYear <- readMaybe birthYearString
            futureYear <- readMaybe futureYearString
            return $ yearDiff futureYear birthYear
    displayAge maybeAge
```

This is very convenient: we've now slurped our two values in our do-notation. If either parse returns Nothing, then the entire do-block will return Nothing. This demonstrates an important property about Maybe: it provides short circuiting.

Without resorting to other helper functions or pattern matching, there's no way to write this code using just fmap. So we've found an example of code that requires more power than Functors provide, and Monads provide that power.

Partial application

But maybe there's something else that provides enough power to write our two-variable code without the full power of `Monad`. To see what this might be, let's look more carefully at our types.

We're working with two values: `readMaybe birthYearString` and `readMaybe futureYearString`. Both of these values have the type `Maybe Integer`. And we want to apply the function `yearDiff`, which has the type `Integer -> Integer -> Integer`.

If we go back to trying to use `fmap`, we'll seemingly run into a bit of a problem. The type of `fmap`- specialized for `Maybe` and `Integer`- is `(Integer -> a) -> Maybe Integer -> Maybe a`. In other words, it takes a function that takes a single argument (an `Integer`) and returns a value of some type `a`, takes a second argument of a `Maybe Integer`, and gives back a value of type `Maybe a`. But our function- `yearDiff`- actually takes two arguments, not one. So `fmap` can't be used at all, right?

Not true. This is where one of Haskell's very powerful features comes into play. Any time we have a function of two arguments, we can also look at is as a function of one argument which returns a function. We can make this more clear with parentheses:

```
yearDiff :: Integer -> Integer -> Integer
yearDiff :: Integer -> (Integer -> Integer)
```

So how does that help us? We can look at the `fmap` function as:

```
fmap :: (Integer -> (Integer -> Integer))
     -> Maybe Integer -> Maybe (Integer -> Integer)
```

Then when we apply `fmap` to `yearDiff`, we end up with:

```
fmap yearDiff :: Maybe Integer -> Maybe (Integer -> Integer)
```

That's pretty cool. We can apply *this* to our `readMaybe futureYearString` and end up with:

```
fmap yearDiff (readMaybe futureYearString) :: Maybe (Integer -> Integer)
```

That's certainly very interesting, but it doesn't help us. We need to somehow apply this value of type `Maybe (Integer -> Integer)` to our `readMaybe birthYearString` of type `Maybe Integer`. We can do this with `do`-notation:

```
#!/usr/bin/env stack
-- stack --resolver lts-7.14 --install-ghc runghc
import Text.Read (readMaybe)

displayAge maybeAge =
    case maybeAge of
        Nothing -> putStrLn "You provided invalid input"
        Just age -> putStrLn $ "In that year, you will be: " ++ show age

yearDiff futureYear birthYear = futureYear - birthYear

main = do
    putStrLn "Please enter your birth year"
    birthYearString <- getLine
    putStrLn "Please enter some year in the future"
    futureYearString <- getLine
    let maybeAge = do
            yearToAge <- fmap yearDiff (readMaybe futureYearString)
            birthYear <- readMaybe birthYearString
            return $ yearToAge birthYear
    displayAge maybeAge
```

We can even use `fmap` twice and avoid the second slurp:

```
#!/usr/bin/env stack
-- stack --resolver lts-7.14 --install-ghc runghc
import Text.Read (readMaybe)

displayAge maybeAge =
    case maybeAge of
        Nothing -> putStrLn "You provided invalid input"
        Just age -> putStrLn $ "In that year, you will be: " ++ show age

yearDiff futureYear birthYear = futureYear - birthYear

main = do
    putStrLn "Please enter your birth year"
    birthYearString <- getLine
    putStrLn "Please enter some year in the future"
    futureYearString <- getLine
    let maybeAge = do
            yearToAge <- fmap yearDiff (readMaybe futureYearString)
            fmap yearToAge (readMaybe birthYearString)
    displayAge maybeAge
```

But we don't have a way to apply our `Maybe (Integer -> Integer)` function to our `Maybe Integer` directly.

Applicative functors

And now we get to our final concept: applicative functors. The idea is simple: we want to be able to apply a function which is *inside* a functor to a value inside a functor. The magic operator for this is `<*>`. Let's see how it works in our example:

```
#!/usr/bin/env stack
-- stack --resolver lts-7.14 --install-ghc runghc
import Text.Read (readMaybe)

displayAge maybeAge =
    case maybeAge of
        Nothing -> putStrLn "You provided invalid input"
        Just age -> putStrLn $ "In that year, you will be: " ++ show age

yearDiff futureYear birthYear = futureYear - birthYear

main = do
    putStrLn "Please enter your birth year"
    birthYearString <- getLine
    putStrLn "Please enter some year in the future"
    futureYearString <- getLine
    let maybeAge =
            fmap yearDiff (readMaybe futureYearString)
                <*> readMaybe birthYearString
    displayAge maybeAge
```

In fact, the combination of `fmap` and `<*>` is so common that we have a special operator, `<$>`, which is a synonym for `fmap`. That means we can make our code just a little prettier:

```
    let maybeAge = yearDiff
            <$> readMaybe futureYearString
            <*> readMaybe birthYearString
```

Notice the distinction between `<$>` and `<*>`. The former uses a function which is *not* wrapped in a functor, while the latter applies a function which is wrapped up.

So we don't need Monads?

So if we can do such great stuff with functors and applicative functors, why do we need monads at all? The terse answer is **context sensitivity**: with a monad, you can make decisions on which processing path to follow based on previous results. With applicative functors, you have to always apply the same functions.

Let's give a contrived example: if the future year is less than the birth year, we'll assume that the user just got confused and entered the values in reverse, so we'll automatically fix it by reversing the arguments to `yearDiff`. With `do`-notation and an if statement, it's easy:

```
#!/usr/bin/env stack
-- stack --resolver lts-7.14 --install-ghc runghc
import Text.Read (readMaybe)

displayAge maybeAge =
    case maybeAge of
        Nothing -> putStrLn "You provided invalid input"
        Just age -> putStrLn $ "In that year, you will be: " ++ show age

yearDiff futureYear birthYear = futureYear - birthYear

main = do
    putStrLn "Please enter your birth year"
    birthYearString <- getLine
    putStrLn "Please enter some year in the future"
    futureYearString <- getLine
    let maybeAge = do
            futureYear <- readMaybe futureYearString
            birthYear <- readMaybe birthYearString
            return $
                if futureYear < birthYear
                    then yearDiff birthYear futureYear
                    else yearDiff futureYear birthYear
    displayAge maybeAge
```

Exercises

1. Implement `fmap` using `<*>` and `return`.

```
#!/usr/bin/env stack
-- stack --resolver lts-7.14 --install-ghc runghc
import Control.Applicative ((<*>), Applicative)
import Prelude (return, Monad)
import qualified Prelude

fmap :: (Applicative m, Monad m) => (a -> b) -> (m a -> m b)
fmap ... ... = FIXME

main =
    case fmap (Prelude.+ 1) (Prelude.Just 2) of
        Prelude.Just 3 -> Prelude.putStrLn "Good job!"
        _ -> Prelude.putStrLn "Try again"
```

Show Solution

2. How is `return` implemented for the `Maybe` monad? Try replacing `return` with its implementation in the code above.

```
#!/usr/bin/env stack
-- stack --resolver lts-7.14 --install-ghc runghc
returnMaybe = FIXME

main
    | returnMaybe "Hello" == Just "Hello" = putStrLn "Correct!"
    | otherwise = putStrLn "Incorrect, please try again"
```

Show Solution

3. yearDiff is really just subtraction. Try to replace the calls to yearDiff with explicit usage of the - operator.

```
#!/usr/bin/env stack
-- stack --resolver lts-7.14 --install-ghc runghc
import Text.Read (readMaybe)

displayAge maybeAge =
    case maybeAge of
        Nothing -> putStrLn "You provided invalid input"
        Just age -> putStrLn $ "In that year, you will be: " ++ show age

main = do
    putStrLn "Please enter your birth year"
    birthYearString <- getLine
    putStrLn "Please enter some year in the future"
    futureYearString <- getLine
    let maybeAge = do
            futureYear <- readMaybe futureYearString
            birthYear <- readMaybe birthYearString
            return $
                -- BEGIN CODE TO MODIFY
                if futureYear < birthYear
                    then yearDiff birthYear futureYear
                    else yearDiff futureYear birthYear
                -- END CODE TO MODIFY
    displayAge maybeAge
```

Show Solution

4. It's possible to write an applicative functor version of the auto-reverse-arguments code by modifying the yearDiff function. Try to do so.

```
#!/usr/bin/env stack
-- stack --resolver lts-7.14 --install-ghc runghc
import Text.Read (readMaybe)
import Control.Applicative ((<$>), (<*>))

displayAge maybeAge =
    case maybeAge of
        Nothing -> putStrLn "You provided invalid input"
        Just age -> putStrLn $ "In that year, you will be: " ++ show age

yearDiff futureYear birthYear = FIXME

main
    | yearDiff 5 6 == 1 = putStrLn "Correct!"
    | otherwise = putStrLn "Please try again"
```

Show Solution

5. Now try to do it without modifying yearDiff directly, but by using a helper function which is applied to yearDiff.

```
#!/usr/bin/env stack
-- stack --resolver lts-7.14 --install-ghc runghc
import Text.Read (readMaybe)
import Control.Applicative ((<$>), (<*>))

displayAge maybeAge =
    case maybeAge of
        Nothing -> putStrLn "You provided invalid input"
        Just age -> putStrLn $ "In that year, you will be: " ++ show age

yearDiff futureYear birthYear = futureYear - birthYear
yourHelperFunction f ...

main
    | yourHelperFunction yearDiff 5 6 == 1 = putStrLn "Correct!"
    | otherwise = putStrLn "Please try again"
```

Show Solution

Read more blog posts

3 Comments **Michael Snoyman's blog** **Disqus' Privacy Policy** 1 **Log**

♡ Recommend Tweet Share Sort by Be

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Glade Diviney • 4 years ago

How is the solution for #4 an applicative functor? It looks like a regular function to me.

^ | ⌄ • Reply • Share ›

Michael Snoyman Mod ➦ Glade Diviney • 4 years ago

The surrounding code in #4 is just a simple test case. The full exercise would be to use this version of yearDiff in the code for the monad section. It may be worth updating the exercise in fact.

^ | ⌄ • Reply • Share ›

Enis Bayramoglu • 4 years ago

Why has the Haskell community forsaken arrows? "The terse answer is context sensitivity: with a monad, you can make decisions on which processing path to follow based on previous results.", but arrows are also powerful enough to express that much. The thing you can't do with an arrow is to construct the rest of the processing based on the previous results.

^ | ⌄ • Reply • Share ›

Subscribe Add Disqus to your siteAdd DisqusAdd ⚠ Do Not Sell My Data

Follow @snoyberg Tweet tc Read more blog posts P Complete I'm a Haskeller