

Programming Paradigms

Unit 14 — Functors and Monads

J. Gamper

Free University of Bozen-Bolzano
Faculty of Computer Science
IDSE

Outline

1 **Functors Revisited**

2 **Monads**

Outline

1 **Functors Revisited**

2 Monads

Beefing Up Functors/1

- What happens if we want to use a function that takes two parameters with a functor?
- For example, lets multiply two values `Just 2` and `Just 5`
`(Just 2) * (Just 5)`
- This does not work, as the multiplication operator `*` expects two numerical values, not two values wrapped in `Maybe`
 - Again: pure function `*`, impure parameters `Just 2` and `Just 5`
- We could push `*` into one of the functors

```
> :t fmap (*) (Just 2)
```

```
fmap (*) (Just 2) :: Num a => Maybe (a -> a)
```

Beefing Up Functors/2

- That means, we now have a function wrapped in a `Just`
- We could also rewrite the above as
`Just (*2)`
- This is a partially evaluated function (remember currying!)
- But we still have a problem: *How do we apply a function that is wrapped inside a functor to values inside a functor box?*
- `fmap` only takes **ordinary functions** and maps them over a functor (box)
 - We saw how to map functions over a `Maybe a`, a list `[a]`, a tree `Tree a`, etc.
- However, `fmap` does **not** work in the following case:
`fmap (Just(*2)) (Just 5)`
- So, what do we do? Rewrite all our multi-parameter functions for functors?

Type Class Applicative

- Not really, there is a **type class** `Applicative` with two important functions

```
pure :: a -> f a
```

```
(<*>) :: f (a -> b) -> f a -> f b
```

- The function `pure` takes a value of any type and returns an **applicative functor** `f` with that value inside it

- i.e., `pure` takes a value and wraps it in an applicative functor box

- The function `<*>`, also called “`ap`” or “`apply`”,

- takes a **functor** `f` that contains a function

- and another **functor that contains a's**, and

- **extracts the function** from the first functor and **maps it** over the second one

- This is exactly what we are looking for, remember:

```
> :t Just (*2)
```

```
Just (*2) :: Num a => Maybe (a -> a)
```

```
> :t Just 5
```

```
Just 5 :: Num a => Maybe a
```

- Compare `(<*>)` to `fmap :: (a -> b) -> f a -> f b`

Using Applicative Functors/1

- Maybe is an instance of Applicative, so we can use its functions right out of the box
 - Well, we have to import the module Control.Applicative first ...

```
> import Control.Applicative
```

```
> (Just (*2)) <*> (Just 5)
```

```
Just 10
```

```
Success!
```

- This also works for values of Nothing

```
> (Just (*2)) <*> Nothing
```

```
Nothing
```

```
> Nothing <*> (Just 5)
```

```
Nothing
```

Using Applicative Functors/2

- As mentioned above, `pure` “wraps” a pure value into an impure context (an applicative functor box)
- We cannot combine pure and impure values in the same computation
- With applicative functors, we wrap the pure value into a (default) impure context:

```
> (Just (*2)) <*> 5
```

does not work

```
> (Just (*2)) <*> (pure 5)
```

```
Just 10
```

does work

Using Applicative Functors/3

- This does not stop at two parameters
- With applicative functors we can chain any number of functors

```
pure f <*> x <*> y <*> z <*> ...
```

- So, for example we define a function summing up three numbers

```
sum3 x y z = x + y + z
```

and then use it in a functor context

```
> pure sum3 <*> Just 4 <*> Just 9 <*> Just 2
Just 15
```

```
> pure sum3 <*> Just 4 <*> Nothing <*> Just 2
Nothing
```

Applicative Instance Implementation for Maybe

- This is how Maybe is defined as an instance of the type class Applicative

```
instance Applicative Maybe where
```

```
  pure = Just
```

```
  Nothing <*> _ = Nothing
```

```
  (Just f) <*> something = fmap f something
```

- The function to wrap a value inside a context is Just (recall that value constructors are functions)
- If the first parameter to (<*>) is Nothing, we cannot extract a function out of it, so the result is Nothing
- If the first parameter is Just with a function f inside, this function is mapped over the second parameter

More Examples of Using Applicative Functors

- Here are some more examples

```
> Just (+3) <*> Just 9  
Just 12
```

```
> pure (+3) <*> Just 10  
Just 13
```

```
> Just (++" world") <*> pure "Hello"  
Hello world
```

```
> Just (++" world") <*> "Hello"  
... error!
```

```
> Just (++" world") <*> Nothing  
Nothing
```

```
> Nothing <*> Just "Hi"  
Nothing
```

- Notice that pure and Just have the same effect here

Outline

1 Functors Revisited

2 **Monads**

Monads

- We will now introduce the concept of **monads** with the help of an example
- Let's assume x persons want to divide up y things:

```
divideUp :: Int -> Int -> Int
divideUp x y = div y x
```

- This is not going to work, as the following **should fail** (but it doesn't)

```
> divideUp 5 12
2
```

Second Try

- We could give back `Maybe Int`
 - If the function fails, we return `Nothing`
 - Otherwise, we return `Just "the result"`

```
divideAmong :: Int -> Int -> Maybe Int
```

```
divideAmong x y =  
  if mod y x /= 0 then  
    Nothing  
  else  
    Just (div y x)
```

```
> divideAmong 5 12
```

```
Nothing
```

```
> divideAmong 6 12
```

```
Just 2
```

- So far, so good

Further Divisions/1

- What happens if we want to divide up one lot among further persons, i.e.:

```
divideAmong 3 (divideAmong 2 12)
```

- This is not going to work, as `divideAmong` expects pure `Ints` as parameters, while it returns a `Maybe` functor (i.e., a `Maybe` box containing the value)
- Let's try using an applicative functor:

```
> pure (divideAmong) <*> Just 2 <*> Just 12  
Just (Just 6)
```

Nope, this adds yet another layer ...

Further Divisions/2

- Is implementing this manually the only option left?

```
divideAmongTwice :: Int -> Int -> Int -> Maybe Int
divideAmongTwice x y z =
  if mod y x /= 0 then
    Nothing
  else
    if mod (div y x) z /= 0 then
      Nothing
    else
      Just (div (div y x) z)
```

- Keeping track of every step that can fail is very awkward and error-prone!
- Monads can help out here

Monads/1

- **Monads** are a **type class** with two important functions:

`return` :: `a -> m a`

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

- The first function **return** wraps a **pure value a** into an impure context, termed a **monad m a**
 - Works like `pure` for applicative functors
- The second function **(>>=)**, called **bind**,
 - takes a **monadic value m a**, i.e., a value of type `a` inside a monadic context
 - and a **function `a -> m b`** that takes a **pure value a** and returns a **monadic value m b**
 - and applies the function to the first parameter (or feeds the parameter into the function), returning a **monadic value m b**
- That is, monads allow **sequenced actions**, i.e., to put together two actions, returning the result of the second one

Monads/2

- Now we can chain together calls of the function

```
> divideAmong 2 120 >>= divideAmong 3 >>= divideAmong 5  
Just 4
```

- And Haskell will keep track of any failures on the way for us

```
> divideAmong 5 12 >>= divideAmong 3 >>= divideAmong 5  
Nothing
```

```
> divideAmong 6 12 >>= divideAmong 3 >>= divideAmong 5  
Nothing
```

Do Notation

- Monads are so important in Haskell that they have their own special notation: the **do notation**
- This notation allows you to chain together monadic function calls in a seemingly imperative way

```
routine :: Maybe Int
routine = do
  x <- divideAmong 2 120
  y <- divideAmong 3 x
  divideAmong 4 y
```

```
routine
Just 5
```

- The statements are executed line by line
- With **<-** we bind a monadic Maybe value (impure) to a variable (pure)
- The result of the final execution is the result of `routine`

IO is a Monad

- Yes, you have seen this notation before in the context of IO
- And, yes, this means that IO is a monad!
- It doesn't end there:
 - There are monads for representing state
 - For dealing with indeterminism
 - Even lists can be interpreted as monads
- There are lots of other things to say about monads
 - All instances of monads need to follow certain laws (instances of (applicative) functors as well)
- But we are going to stop here

Mathematical Foundation

- The concepts used in Haskell did not just fall from the sky
- They are rooted in mathematical theory, **category theory** to be more specific
- In category theory, mathematicians try to capture the underlying properties of mathematical concepts
- Expressed in simplified terms, it is like finding and defining “type classes” for mathematical structures

Summary – Strengths of Haskell

- The **type system (strong/static)** prevents you from making a lot of mistakes
 - Nevertheless, it is quite **flexible** when it comes to extending it with **user-defined types**
- Haskell offers a lot in terms of **expressiveness**, yielding very **concise code**
- Haskell is a **pure** functional language, providing **referential transparency**
 - functions give the same output for the same input
 - functions have no side effects
 - a variable can only be assigned a value once
- Haskell uses **curried functions** in combination with **partial evaluation** of functions,
 - i.e., internally, functions have only one input parameter;
 - functions with multiple input parameters are decomposed into a sequence of partial functions, each having one parameter
- It is easier to show the **correctness** of your programs, due to the **pure functional** style
- It does **lazy evaluation**, which gives you an additional tool for writing programs efficiently
- It supports **list comprehension** and **infinite lists**

Summary – Weaknesses of Haskell

- The pure functional paradigm also has a price: dealing with **messy real-world situations** such as IO and state is not easy
- Haskell has a **steep learning curve**, i.e., it takes a while to learn how to wield the power of Haskell
- This may also explain the fact that the Haskell community is relatively small