The PHP Foundation

Monad for Dummies

Gina Peter Banyard

March 15, 2024

About me

Website https://gpb.moe
Mastodon `@Girgias@phpc.social`

- Studied pure mathematics at Imperial College London
- PHP Core developer financed by The PHP Foundation
- Lead maintainer for the French documentation of PHP
- Cares about type systems
- Cares about PHP's semantics

The essence of functional programming

Pure functions are functions that:

- have no state
- do not cause side effects
- produces the same result given the same inputs

```
function add(int $a, int $b): int {
  return $a + $b;
}
```

Some impure functions: `fgets()`, `array_rand()`, `sort()`

The essence of functional programming

Values are immutable.

Therefore, loops are not possible. Recursive functions are used instead:

```
function get_product_price($products, int $index = 0) {
  if (count($products) >= $index) {
    return [];
  }
  $prices = get_product_price($products, $index + 1);
  $price = $product[$index]->price;
  return [$price, ...$prices];
}
```

The essence of functional programming

Instead of attaching functions to data (i.e. methods of a class).

Data is attached to functions. Partial application of functions, is the Dependency Injection concept for functions:

The essence of functional programming

Instead of attaching functions to data (i.e. methods of a class).

Data is attached to functions. Partial application of functions, is the Dependency Injection concept for functions:

```
$add2 = fn (int $a) => add($a, 2);

$findDpcAttendees = fn ($attendees)
  => filterAttendeesByConference($attendees, 'dpc');
```

In an ideal world

Piping functions together to process our data:

```
$user = get_user_via_id($id);
$cart = get_user_cart($user);
$products = get_products_from_cart($cart);
$prices = get_product_price($products);
$total_price = array_sum($prices);
charge_amount($total_price);
dispatch_products($products);
```

In the real world

```php
$user = get_user_via_id($id);
if ($user === null) { /* Error handling */ }
$cart = get_user_cart($user);
try {
  $products = get_products_from_cart($cart);
} catch (\ProductSoldOut $e) { /* Error handling */ }
$prices = get_product_price($products);
$total_price = array_sum($prices);
try {
  charge_amount($total_price);
} catch (\PaymentFailure $e) { /* Error handling */ }
dispatch_products($products); // Write to DB/File
```

How to have **state** when pure functions are stateless and values immutable?

How to have **state** when pure functions are stateless and values immutable?

By using a *monad*!

What is a Monad?

In category theory, a branch of mathematics, a **monad** is a *monoid* in the *category* of *endofunctors* of some fixed category. [...] a monad is an *endofunctor* together with two *natural transformations*.

— Wikipedia “Monad (category theory)”

What is a Monad, in programming?

In functional programming, a **monad** is a structure that combines *functions* and wraps their return values in a *type* with additional computation. [...] Monads define two *operators*: one to wrap a value in the monad type, and another to compose together functions that output values of the monad type.

— Wikipedia “Monad (functional programming)”

What is a Monad, in practice?

A **monad** M is a wrapper around some *data* of type T, written as $M\langle T \rangle$ which has 2 associated functions:

1. A *unit* or *return* function: `function unit(T $v): M<T>`
2. A *bind* function: `function bind(callable $fn): M<S>`, where `$fn` is a function with signature `function(T $v): M<S>`

The ubiquitous monad example: Maybe Monad

```
final class Maybe {
  public readonly mixed $data;
  public static function unit(mixed $data): Maybe {
    $o = new Maybe; $o->data = $data; return $o;
  }
  public function isNothing(): bool {
      return !array_key_exists('data', get_object_vars($this));
  }
  public function bind(callable $fn): Maybe {
    return $this->isNothing() ? $this : $fn($this->data);
  }
}
```

The Maybe Monad in action

```
final class Product {
    public function __construct(public int $price) {}
}
function get_product(int $id) {
    return $id%2 ? Maybe::unit(new Product(20)) : new Maybe;
}
function get_price(Product $p): int { return $p->price; }
function apply_taxes(int $i): int { return intdiv($i*120, 100); }
function maybify_pure_result(callable $fn): Closure {
  return fn ($input) => Maybe::unit($fn($input));
}
```

The Maybe Monad in action

```
$m = get_product($id);
$r = $m->bind(maybify_pure_result(get_price(...)));
$r = $r->bind(maybify_pure_result(apply_taxes(...)));
var_dump($r->isNothing() ? "Nothing" : $r->data);
```

Prints:

- `string(7) "Nothing"` if the ID is even
- `int(24)` if the ID is odd

But why not?

```
final class Product {
    public function __construct(public int $price) {}
    public function applyTaxes(): self { $this->price *= 1.2; }
}
final class Db {
  public function getProductById(int $id): ?Product
}
$finalPrice = $db->getProductById($id)?->applyTaxes()->price;
```

The Either Monad

The *Either* monad $E\langle L, R\rangle$ wraps 2 values instead of a single one:

- Left: which holds a value of type L
- Right: which holds a value of type R

We have 2 *units*, one for each case.

The *bind* function will unwrap the R value and pass it to the given function if it set. Otherwise, L is returned.

$$f : R_1 \to E\langle L_2, R_2\rangle$$

$$b(f) : E\langle L_1, R_1\rangle \to E\langle L_1 | L_2, R_2\rangle$$

The Either Monad

```php
final class Either {
    public readonly mixed $left; // Error
    public readonly mixed $right; // Ok
    public static function left(mixed $data): Either {
      $o = new Either; $o->left = $data; return $o;
    }
    public static function right(mixed $data): Either {
      $o = new Either; $o->right = $data; return $o;
    }
    public function isRight(): bool {
        return array_key_exists('right', get_object_vars($this));
    }
    public function bind(callable $fn): Either {
      return $this->isRight() ? $fn($this->right) : $this;
    }
}
```

Either monad in action

```
class File {}

enum OpenFileErrors {
    case FileDoesNotExist;
    case AccessDenied;
    case IsDirectory;
}

/** @return Either<OpenFileErrors, File> */
function open_file(string $path): Either {
  return Either::left(OpenFileErrors::FileDoesNotExist);
  return Either::right(new File());
}
```

Either monad in action

```
enum GetContentErrors {
    case FileNotReadable;
}

/** @return Either<GetContentErrors, string> */
function get_content_file(File $f): Either {
  return Either::left(GetContentErrors::FileNotReadable);
  return Either::right("Content of the file");
}
```

Either monad in action

```
final class TabularData {}
enum ParseToCsvErrors {
    case UnexpectedEof;
    case InconsistentLineFields;
}

/** @return Either<ParseToCsvErrors, TabularData> */
function parse_content_into_tabular(string $content): Either {
  return Either::left(ParseToCsvErrors::InconsistentLineFields);
  return Either::right(new TabularData());
}
```

Either monad in action

```
$data = open_file('')
    ->bind(get_content_file(...))
    ->bind(parse_content_into_tabular(...));
if ($data->isRight()) {
  /* Do some more stuff with tabular data */
} else {
  $errorCode = match ($data->left) {
    OpenFileErrors::FileDoesNotExist => 1,
    OpenFileErrors::AccessDenied => 2,
    OpenFileErrors::IsDirectory => 3,
    GetContentErrors::FileNotReadable => 4,
    ParseToCsvErrors::UnexpectedEof => 5,
    ParseToCsvErrors::InconsistentLineFields => 6,
  };
}
```

The Either monad: generics pushed to the limits

The Either monad wraps a union type of 2 types.

The Maybe monad can be viewed as a specialised Either monad. The type of L is always `null`.

The Exception monad is similar, but the type of L is always a `string`

Monads are not *just* about error handling!

The List monad

Lists are monads when specific functions are associated to them!

The List monad is also great for deriving the monad from first principal from the category theory point of view!

Because List is a functor, as it has a `map()` function!

The 2 natural transformations:

- A unit function $\eta : T \to M\langle T \rangle$
- A join function $\mu : M\langle M\langle T \rangle\rangle \to M\langle T \rangle$

The List monad

Map: `array_map()`.

Unit: `return [$value];`

Join:

```
function concat_lists(array $a) {
  return array_merge(...$a);
}
```

And we can derive our `bind()` function:

```
function bind(callable $fn): Closure {
  return fn (array $a) => concat_lists(array_map($fn, $a));
}
```

The List monad in action

```
function get_recomended_products(string $product) {
    return [
        'fancy ' . $product,
        'budget ' . $product,
        'brandA ' . $product,
    ];
}

$products = ['shampoo', 'cereals', 'book'];

$r = bind(get_recomended_products(...))($products);
var_dump($r);
```

The List monad in action

```
array(9) {
  [0]=> string(13) "fancy shampoo"
  [1]=> string(14) "budget shampoo"
  [2]=> string(14) "brandA shampoo"
  [3]=> string(13) "fancy cereals"
  [4]=> string(14) "budget cereals"
  [5]=> string(14) "brandA cereals"
  [6]=> string(10) "fancy book"
  [7]=> string(11) "budget book"
  [8]=> string(11) "brandA book"
}
```

Warning

The following monad will be written with visible-side effects

The Logger monad

Also known as the **Writer** or **Action** monad.

```
final class LoggerMonad {
    public function __construct(public mixed $data,
      public array $logs = []) {}
    public function bind(callable $fn) {
        $resultLoggerMonad = $fn($this->data);
        return new LoggerMonad(
            $resultLoggerMonad->data,
            [...$this->logs, ...$resultLoggerMonad->logs],
        );
    }
}
```

The Logger monad

```
function loggify(callable $fn): Closure {
  return function ($value) use ($fn) {
    $name = (new ReflectionFunction($fn))->name;
    $log = [
      'Running ' . $name . '('
      . var_export($value, true) .')'
    ];
    return new LoggerMonad($fn($value), $log);
  };
}
```

The Logger monad in action

```
function log_calls($value, callable ...$fns) {
    $logging_fns = array_map(loggify(...), $fns);
    $monad = new LoggerMonad($value);
    foreach ($logging_fns as $fn) {
        $monad = $monad->bind($fn);
    }
    return $monad;
}

function add2($v) { return $v + 2; }
function square($v) { return $v*$v; }
function mul3($v) { return $v*3; }
function div6($v) { return $v/6; }
```

The Logger monad in action

```
$m = log_calls(
    3,
    add2(...),
    square(...),
    mul3(...),
    div6(...),
);

echo 'Value is:'.$m->data."\n"
  .join($m->logs, "\n");
```

```
Value is:12.5
Running add2(3)
Running square(5)
Running mul3(25)
Running div6(75)
```

Thank you!