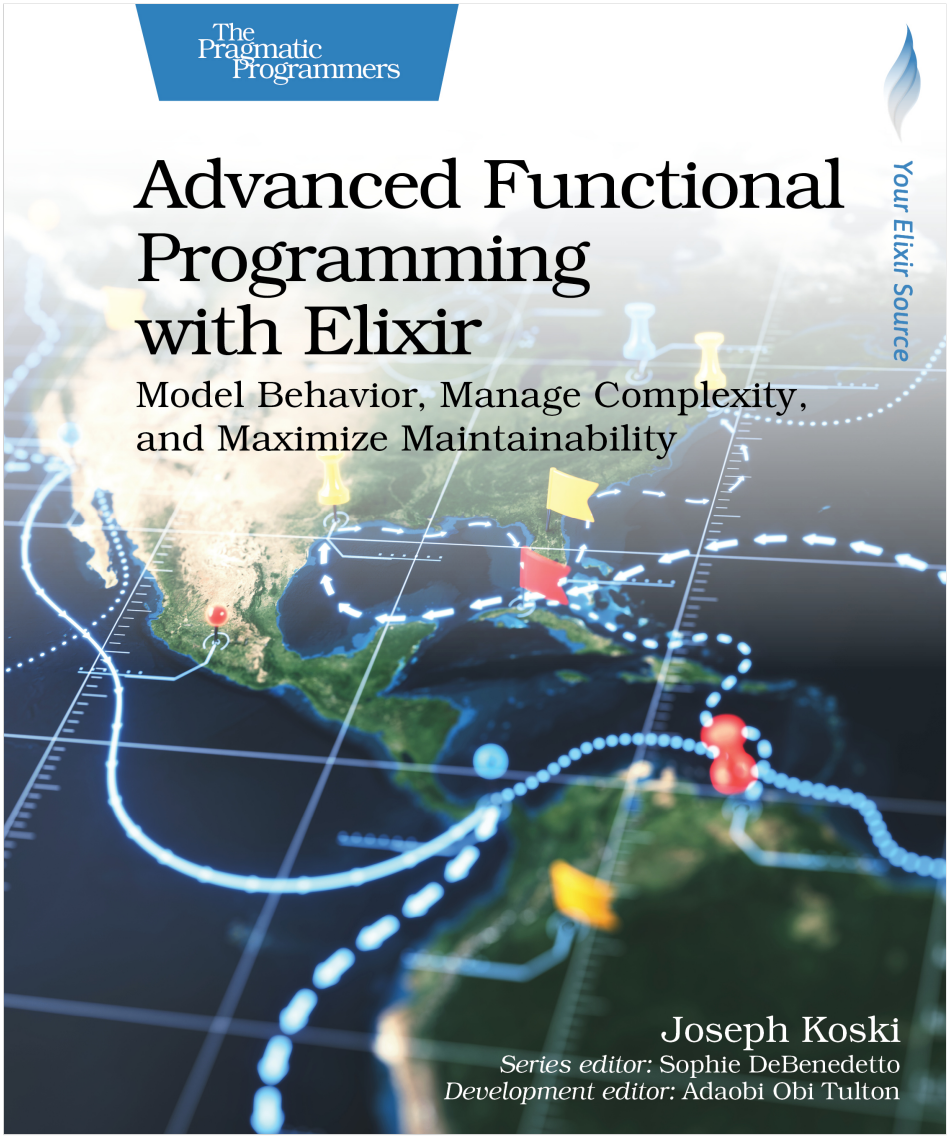

This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

For more information, or to purchase a paperback or ebook copy, please visit https://www.pragprog.com.

Build the Monad

In his 1995 paper *Monads for Functional Programming*, Philip Wadler outlined two essential operations that comprise a monad: one to applies a transformation while preserving structure (map), and one chains context-aware computations (bind).[1] Today, most definitions of a monad support more than just these two—but they remain the foundation.

Transform with a Functor

Anyone who has mapped over a list has used a functor. Each item is transformed, but the structure stays the same—returning the same number of items, in the same order.

More specifically, a functor follows two rules:

- Identity: Mapping with the identity function returns a copy the original structure. map(fn x -> x end, F(a)) = F(a)
- Composition: Mapping in two steps is the same as mapping once with a composed function. map(f, map(g, F(a))) = map(fn x -> f.(g.(x)) end, F(a))

Let's see Elixir's Enum functor in action.

Our Patron expert introduces a new promotion that adds 10 points to a patron's reward_points.

lib/fun_park/patron.ex

```
def promotion(%__MODULE__{} = patron, points) do
  new_points = Math.sum(get_reward_points(patron), points)

  change(patron, %{reward_points: new_points})
end
```

This uses change/2 to apply a promotion by incrementing a patron's existing reward points.

Run It

Start Elixir's iex shell and run the mix script to compile the code and load the project.

```
iex -S mix
```

Let's start by generating a list of patrons:

```
iex> alice = FunPark.Patron.make("Alice", 14, 125, reward_points: 25)
```

1. https://homepages.inf.ed.ac.uk/wadler/papers/marktoberdorf/baastad.pdf

```
iex> beth = FunPark.Patron.make("Beth", 15, 140, reward_points: 10)
iex> charles = FunPark.Patron.make("Charles", 13, 130, reward_points: 50)
iex> patrons = [alice, beth, charles]
```

With Elixir's Enum.map/2 functor, we can apply promotion/2 to each patron, adding 10 to their reward points.

```
iex> patrons |> Enum.map(&FunPark.Patron.promotion(&1, 10))
[
  %FunPark.Patron{name: "Alice", reward_points: 35, ...},
  %FunPark.Patron{name: "Beth", reward_points: 20, ...},
  %FunPark.Patron{name: "Charles", reward_points: 60, ...}
]
```

The functor applies the transformation to each element, preserving the list's structure—its length and order remain unchanged.

Sequence Computations

A monad includes behavior for chaining computations within a context. But unlike map, there's no universally agreed-upon name for this operation, so we'll follow Haskell's convention and call it bind.

The bind operation follows three laws:

- Left Identity: Wrapping a value and then binding it to a function is the same as applying the function directly. bind(pure(a), f) = f(a)
- Right Identity: Binding a monad to pure has no effect. bind(m, pure) = m
- Associativity: It doesn't matter how you nest your bindings—the result is the same. bind(bind(m, f), g) = bind(m, fn x -> bind(f(x), g) end)

These laws describe how bind behaves, but you don't need to memorize them. Their purpose is to ensure that chaining behaves predictably.

Elixir's Enum includes bind, but under the name flat_map/2.

Run It

First, we need to define a *Kleisli function*—named after Heinrich Kleisli—it is a function that takes an input and returns a monad.

```
kleisli_fn = fn x -> if rem(x, 2) == 0, do: [x * x], else: [] end
```

This Kleisli function takes a number and returns a list: the squared value for even numbers, or an empty list for odd ones.

Next, we need a list of values:

```
list = [1, 2, 3, 4, 5, 6]
```

When we apply our Kleisli function:

```
list |> Enum.flat_map(kleisli_fn)
[4, 16, 36]
```

We get a new list. Unlike map, bind allowed us to reshape the structure—we started with six items and ended with three.

map and bind are both context-aware—in this case, the context is a list. map transforms each item while preserving the structure; bind allows the structure to change.

Interestingly, the monad is not a static concept. Most modern formulations include a third block of logic called the applicative.[2]

Independent Computations

The Applicative is useful when we need to combine two things that are already inside a context. It follows four fundamental rules:

- Identity: Applying a wrapped identity function has no effect. ap(pure(fn x -> x end), F(a)) = F(a)
- Homomorphism: Lifting a function and a value separately is the same as applying them directly. ap(pure(f), pure(a)) = pure(f.(a))
- Interchange: A function in context can be applied to a pure value—or the value can be lifted into a function and applied to the context instead. ap(F(f), pure(a)) = ap(pure(fn g -> g.(a) end), F(f))
- Composition: Applying functions step by step inside the context behaves the same as applying them all at once. ap(ap(ap(pure(fn f -> fn g -> fn x -> f.(g.(x))) end, F(f)), F(g)), F(a)) = ap(F(f), ap(F(g), F(a)))

Uffda… that's a lot. But here's the heart of it: these rules make sure that applying functions in a context behaves just like it would outside the context.

Unlike map, or bind there is no applicative function in Elixir's Enum module.

However, we can make one:

Run It

```
ap = fn values, funcs -> for f <- funcs, v <- values, do: f.(v) end
```

2. https://www.staff.city.ac.uk/~ross/papers/Applicative.html

Here, our ap/2 function works within the context of lists, taking a list of values and a list of functions, and applying each function to every value—producing a new list of results.

Next, we need a couple of simple functions, add_one/1 and add_two/1:

```
add_one = fn x -> x + 1 end
add_two = fn x -> x + 2 end
func_list = [add_one, add_two]
```

We have assembled our functions into a list of functions.

And we need a list of values:

```
list = [10, 20, 30]
```

Finally, we use ap to apply our list of functions to our list of values:

```
list |> ap.(func_list)
[11, 21, 31, 12, 22, 32]
```

The result is a list of all calculations. Like bind, ap operates within a context—and can reshape the structure. Here, the output list is longer than the input because each function is applied to every value.

With bind, each step depends on the result of the previous one. With ap, each function is applied *independently* to each input, with no dependency between steps.

map transforms. bind chains. ap collects.