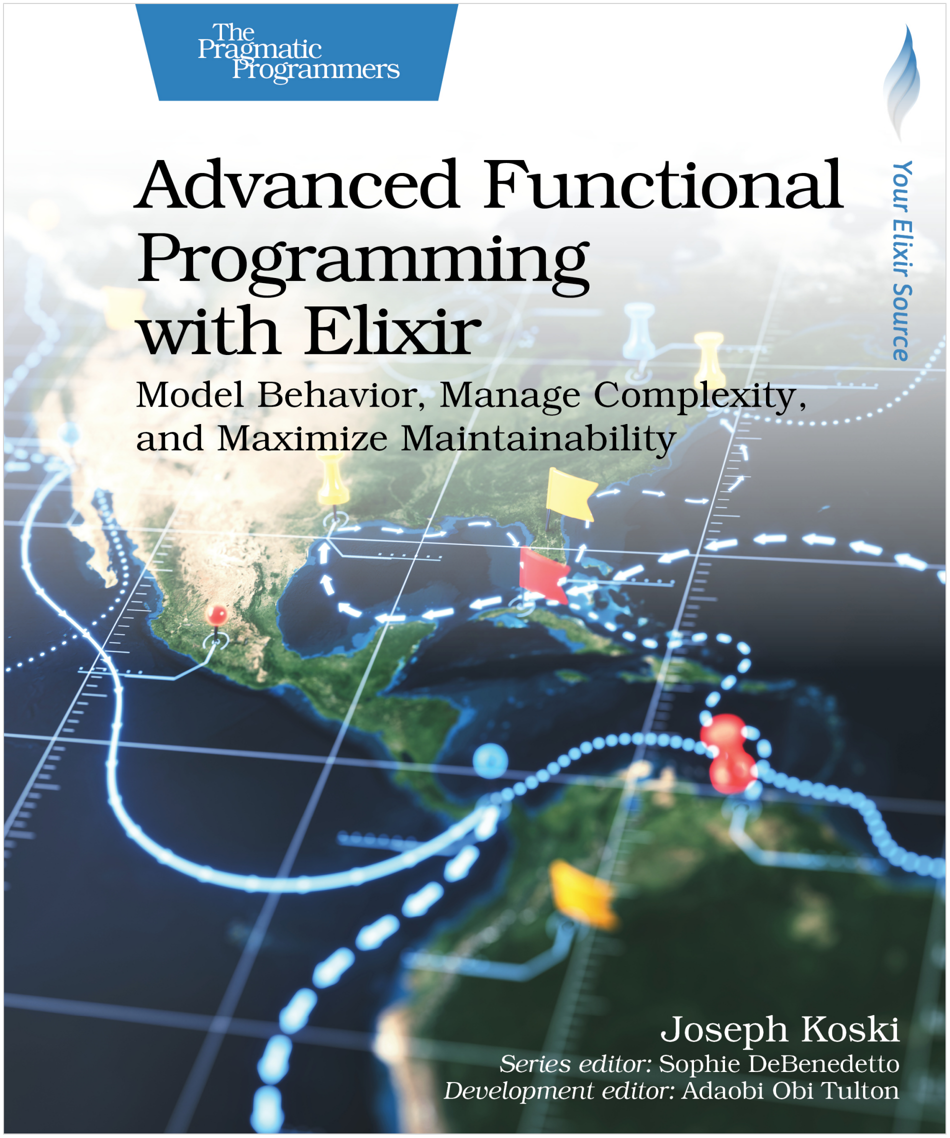
The Pragmatic Programmers

Advanced Functional Programming with Elixir

Model Behavior, Manage Complexity, and Maximize Maintainability

Your Elixir Source

Joseph Koski

Series editor: Sophie DeBenedetto

Development editor: Adaobi Obi Tulton

Any

The Predicate.Any monoid combines predicates in the context of disjunction (∨), using an identity of () -> false.

lib/fun_park/monoid/pred_any.ex

```
defmodule FunPark.Monoid.Predicate.Any do
  defstruct value: &FunPark.Monoid.Predicate.Any.default_pred?/1

  def default_pred?(_), do: false
end

defimpl FunPark.Monoid, for: FunPark.Monoid.Predicate.Any do
  alias FunPark.Monoid.Predicate.Any

  def empty(_), do: %Any{}

  def append(%Any{} = p1, %Any{} = p2) do
    %Any{
      value: fn value -> p1.value.(value) or p2.value.(value) end
    }
  end

  def wrap(%Any{}, value) when is_function(value, 1) do
    %Any{value: value}
  end

  def unwrap(%Any{value: value}), do: value
end
```

The best part about monoids is that they're closed under their operation—combine two, and you get another of the same kind. With predicates, every composition returns a predicate.

As in earlier chapters, let's abstract away the monoids and give callers higher-level operations instead:

lib/fun_park/predicate.ex

```
defmodule FunPark.Predicate do
  import FunPark.Monoid.Utils, only: [m_append: 3, m_concat: 2]
  alias FunPark.Monoid.Predicate.{All, Any}

  def p_and(pred1, pred2) when is_function(pred1) and is_function(pred2) do
    m_append(%All{}, pred1, pred2)
  end

  def p_or(pred1, pred2) when is_function(pred1) and is_function(pred2) do
    m_append(%Any{}, pred1, pred2)
  end

  def p_not(pred) when is_function(pred) do
    fn value -> not pred.(value) end
  end
  def p_all(p_list) when is_list(p_list) do
    m_concat(%All{}, p_list)
  end

  def p_any(p_list) when is_list(p_list) do
    m_concat(%Any{}, p_list)
  end

  def p_none(p_list) when is_list(p_list) do
    p_not(p_any(p_list))
  end

end
```

We prefix these functions with `p_`, avoiding conflicts with Elixir's reserved words like `and` and `or`.

Now we can implement `suggested?/1` by composing `online?/1` with the negation of `long_wait?/1`, ensuring a ride is only suggested if it's available and the wait is short:

lib/fun_park/ride.ex

```
def suggested?(%__MODULE__{} = ride),
  do: p_all([&online?/1, p_not(&long_wait?/1)]).(ride)
```

There's more than meets the eye in this simple function. Monoids can combine concrete values—like numbers using `max` or `sum`. But in this case, we're combining *ideas*. A predicate isn't a boolean—it's a function that returns a boolean.

The `suggested?/1` function is just a wrapper—it takes a ride and immediately applies it to the predicate. In theory, we should be able to eliminate the wrapper with eta reduction. However, Elixir requires anonymous functions to be called explicitly with `.()` syntax. Because the outer function is named and the inner one is anonymous, we can't reduce it.

Run It

The Tea Cup ride is not suggested because it has a wait time of 100 minutes:

```
iex> tea_cup = FunPark.Ride.make("Tea Cup", online: true, wait_time: 100)
%FunPark.Ride{ name: "Tea Cup", wait_time: 100, online: true, ...}

iex> FunPark.Ride.suggested?(tea_cup)
false
```

Later, the wait time for the Tea Cup ride shortens to 10 minutes, making it a suggested ride:

```
iex> tea_cup = FunPark.Ride.change(tea_cup, %{wait_time: 10})
%FunPark.Ride{ name: "Tea Cup", wait_time: 10, online: true, ... }

iex> FunPark.Ride.suggested?(tea_cup)
true
```

But our Ride expert is back—we're suggesting rides to patrons who aren't eligible! It won't do for patrons to follow our suggestion only to find out they can't take the ride.

So far, we've operating within a single bounded context. But with this request, our expert needs us to span both Ride and Patron.

Predicates That Span Contexts

In *Domain-Driven Design [Eva03]*, when bounded contexts interact, their relationship is defined via *context mapping*. In FunPark, the Ride context sets eligibility rules—such as height and age requirements—while the Patron context supplies height and age attributes. This forms a *conformist relationship*, where Patron conforms to the rules set by Ride but has no influence over them.

Since the Ride defines the rules, the logic for determining eligibility belongs in the Ride bounded context.

To determine eligibility, a Ride must verify that a Patron meets height *and* age requirements.

First, let's define predicates for height and age requirements in the Ride context:

lib/fun_park/ride.ex
```
def tall_enough?(%Patron{} = patron, %__MODULE__{min_height: min_height}),
  do: Patron.get_height(patron) >= min_height
```

lib/fun_park/ride.ex
```
def old_enough?(%Patron{} = patron, %__MODULE__{min_age: min_age}),
  do: Patron.get_age(patron) >= min_age
```

Notice that we're not destructuring Patron to access age or height. How that data is stored is an implementation detail. The Ride context shouldn't rely on it—it should use Patron's accessors instead. This keeps the contexts loosely coupled and allows Patron to evolve without breaking Ride.

Next, we define eligible?/2 by combining the predicates using p_all/1:

```
def eligible?(%Patron{} = patron, %__MODULE__{} = ride),
  do: p_all([&tall_enough?/2, &old_enough?/2]).(patron, ride)
```

Run It

Let’s start with a patron and a ride:

```
iex> roller_mtn = FunPark.Ride.make(
  "Roller Mountain", min_height: 120, min_age: 12
)
%FunPark.Ride{ name: "Roller Mountain", min_age: 12, min_height: 120, ... }

iex> alice = FunPark.Patron.make("Alice", 13, 119)
%FunPark.Patron{ name: "Alice", age: 13, height: 119, ...}
```

Alice meets the age requirement but does not meet the height requirement:

```
iex> alice |> FunPark.Ride.old_enough?(roller_mtn)
true

iex> alice |> FunPark.Ride.tall_enough?(roller_mtn)
false
```

This means Alice is not eligible to ride Roller Mountain:

```
iex> alice |> FunPark.Ride.eligible?(roller_mtn)
false
```

However, if Alice grows a bit over the summer, she will be eligible:

```
iex> alice = FunPark.Patron.change(alice, %{height: 121})
%FunPark.Patron{ name: "Alice", age: 13, height: 121 }

iex> alice |> FunPark.Ride.eligible?(roller_mtn)
true
```

By composing logic across contexts, we’re no longer suggesting rides a patron can’t take.

We’ve enforced a rule across two contexts—without entangling them.