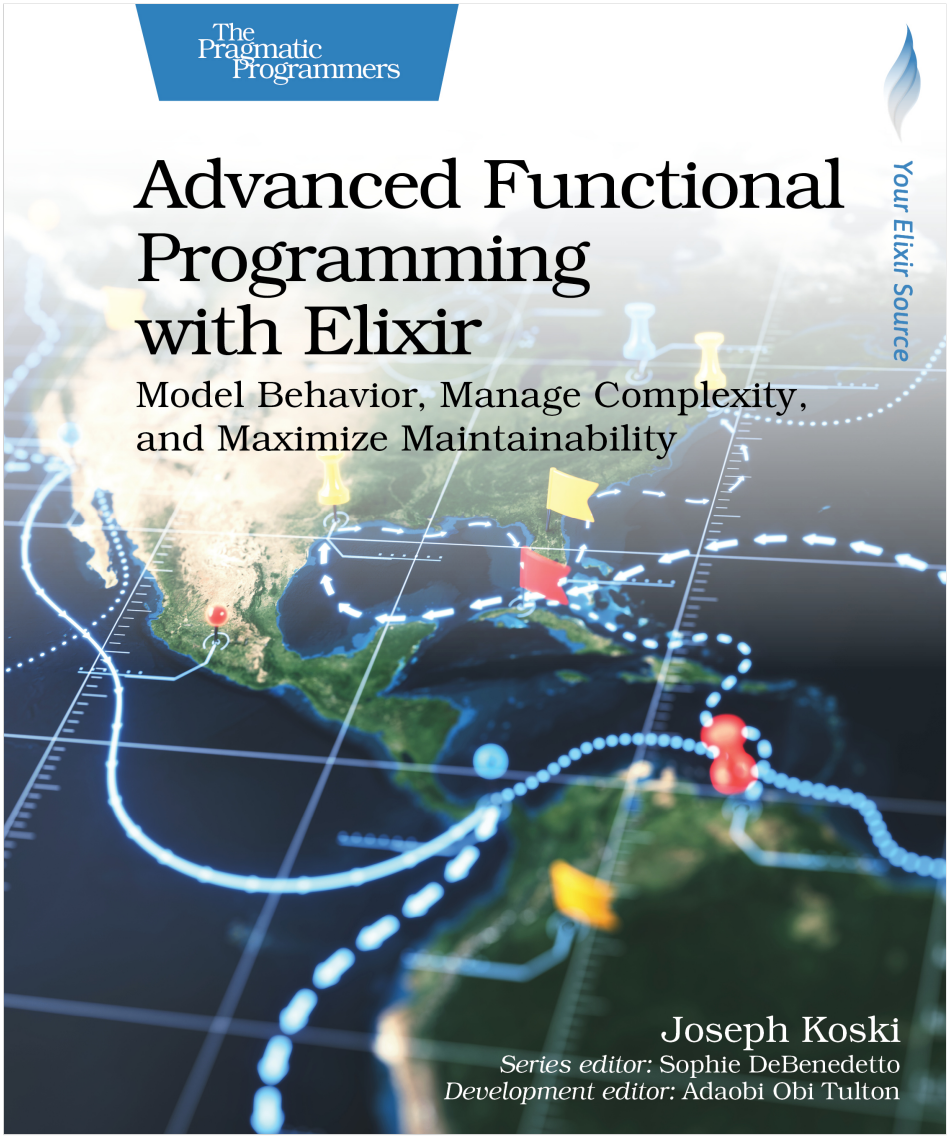

This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

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

Equality Is Contextual

No problem, we can just adjust the fast pass implementation to target the time rather than the ID.

```
defimpl FunPark.Eq, for: FunPark.FastPass do
  alias FunPark.Eq
  alias FunPark.FastPass
  def eq?(%FastPass{time: v1}, %FastPass{time: v2}), do: Eq.eq?(v1, v2)

  def not_eq?(%FastPass{time: v1}, %FastPass{time: v2}),
    do: Eq.not_eq?(v1, v2)
end
```

But wait, we don't want to replace the default `Eq` for `FastPass`. Instead, we need an alternative `Eq` that considers two fast passes equal if they occur at the same time.

Transform Inputs Before Matching

If you've worked with lists, you're likely familiar with `map`, an implementation of a *functor*, which applies a function to transform something while preserving its structure—for a list, this means transforming each item. Functional programming also includes the *contravariant functor*, which transforms the input before it is processed. This is useful for abstractions like `Eq` and `Ord`, where we often want to transform a value before it enters the comparison—such as extracting a specific field or normalizing the input to ignore differences like case or formatting.

We'll cover functors in more detail: Transform with a Functor, on page ?

Unlike some other functional languages, Elixir doesn't have a built-in `contramap` function, so we need to implement it ourselves.

```
defmodule FunPark.Eq.Utils do
  alias FunPark.Eq

  def contramap(f) do
    %{
      eq?: fn a, b -> Eq.eq?(f.(a), f.(b)) end,
      not_eq?: fn a, b -> Eq.not_eq?(f.(a), f.(b)) end
    }
  end
end
```

The `contramap/1` function is a *higher-order transformer:* it takes an existing comparator and adapts it to work on a different type by applying a function *before* comparing values.

In our FastPass module, we define get_time/1 to extract the time value from a fast pass and implement eq_time/0 using contramap/1, establishing equality based on time.

lib/fun_park/fast_pass.ex

```
def get_time(%__MODULE__{time: time}), do: time

def eq_time do
  Eq.Utils.contramap(&get_time/1)
end
```

Now, if we want to compare two FastPass values by time, we can use eq_time.

Run It

Create the rides:

```
iex> mansion = FunPark.Ride.make("Dark Mansion", min_age: 14, tags: [:dark])
iex> tea_cup = FunPark.Ride.make("Tea Cup")
```

Generate a fast pass for the Dark Mansion:

```
iex> datetime = DateTime.new!(~D[2025-06-01], ~T[13:00:00])

iex> fast_pass_a = FunPark.FastPass.make(mansion, datetime)
%FunPark.FastPass{
  id: 3618,
  ride: %FunPark.Ride{
    id: 3490,
    name: "Dark Mansion",
    ...
  },
  time: ~U[2025-06-01 13:00:00Z]
}
```

Generate another Fun Pass for the Tea Cup using the same time:

```
iex> fast_pass_b = FunPark.FastPass.make(tea_cup, datetime)
%FunPark.FastPass{
  id: 3650,
  ride: %FunPark.Ride{
    id: 3522,
    name: "Tea Cup",
    ...
  },
  time: ~U[2025-06-01 13:00:00Z]
}
```

Our default equality check knows these are different (they have different ids):

```
iex> FunPark.Eq.eq?(fast_pass_a, fast_pass_b)
false
```

But our new custom equality knows they have the same time:

```
iex> FunPark.FastPass.eq_time.eq?.(fast_pass_a, fast_pass_b)
true
```

You're Gonna Need a Bigger Boat

Sometimes, our tools aren't big enough to handle what's lurking beneath.

Let's take a closer look at our current `contramap/1` function:

```
def contramap(f) do
  %{
    eq?: fn a, b -> Eq.eq?(f.(a), f.(b)) end,
    not_eq?: fn a, b -> Eq.not_eq?(f.(a), f.(b)) end
  }
end
```

In Elixir, protocols are tied to named modules. Our custom `Eq` instance, derived in `contramap/1`, isn't associated with any particular module, so we store its comparison functions in a map.

Because Elixir calls functions differently depending on where they're stored—implicitly for modules and explicitly for maps—we can't treat them interchangeably. To keep things composable and consistent, our best option is to normalize everything to maps.

lib/fun_park/eq/utils.ex

```
def to_eq_map(%{eq?: eq_fun, not_eq?: not_eq_fun} = eq_map)
    when is_function(eq_fun, 2) and is_function(not_eq_fun, 2) do
  eq_map
end

def to_eq_map(module) when is_atom(module) do
  %{
    eq?: &module.eq?/2,
    not_eq?: &module.not_eq?/2
  }
end
```

The `to_eq_map/1` function ensures that we always work with a map representation of equality checks. If we pass a map that has `eq?` and `not_eq?` functions, it is returned as-is. If we pass in a module, it extracts the corresponding functions and wraps them in a map, standardizing our interface.

Here is an improved `contramap/1` function using this approach:

lib/fun_park/eq/utils.ex

```
def contramap(f, eq \\ Eq) do
  eq = to_eq_map(eq)

  %{
    eq?: fn a, b -> eq.eq?.(f.(a), f.(b)) end,
    not_eq?: fn a, b -> eq.not_eq?.(f.(a), f.(b)) end
  }
end
```

Not only have we standardized the representation, but `eq \\ Eq` preserves the default while still allowing us to swap in custom equality logic.

This is a bigger—much more composable—boat.

Simplify Equality Checks

Even though we've standardized our internal logic around the map-based representation of `Eq`, usage still differs depending on whether it's produced by the protocol or by `contramap/2`. From the caller's perspective, that distinction is an internal detail. Let's introduce a single `eq?/2` function to fold that difference into a unified interface.

lib/fun_park/eq/utils.ex

```
def eq?(a, b, eq \\ Eq) do
  eq = to_eq_map(eq)
  eq.eq?.(a, b)
end
```

Now, callers have a single entry point that uses the protocol by default but still allows custom equality logic to be passed in as an optional parameter. This keeps comparison logic adaptable without changing the underlying data, helping the system stay flexible as it evolves.

Run It

Let's regenerate our passes:

```
iex> mansion = FunPark.Ride.make("Dark Mansion", min_age: 14, tags: [:dark])
iex> tea_cup = FunPark.Ride.make("Tea Cup")

iex> datetime = DateTime.new!(~D[2025-06-01], ~T[13:00:00])

iex> fast_pass_a = FunPark.FastPass.make(mansion, datetime)
iex> fast_pass_b = FunPark.FastPass.make(tea_cup, datetime)
```

From the context of `FastPass`, the passes have different ID's, so they are different:

```
iex> FunPark.Eq.Utils.eq?(fast_pass_a, fast_pass_b)
false
```

But we can inject our eq_time/0 logic to determine they are scheduled for the same time:

```
iex> has_eq_time = FunPark.FastPass.eq_time()
iex> FunPark.Eq.Utils.eq?(fast_pass_a, fast_pass_b, has_eq_time)
true
```

We've structured equality around domain context—using protocols for shared behavior and contramap to adapt comparisons to specific needs, making the system easier to use and evolve.