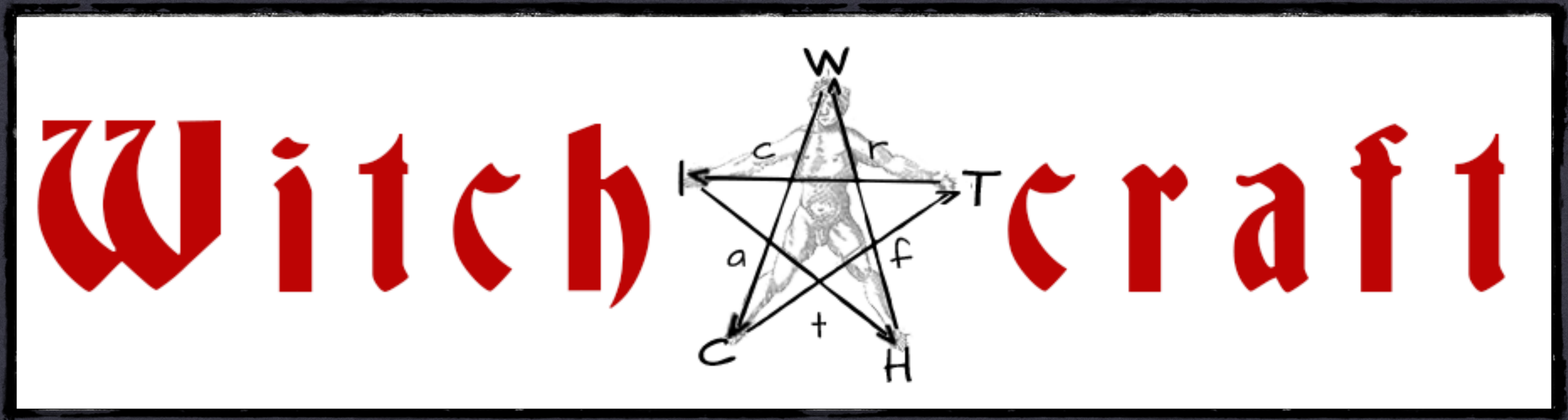


WITCHCRAFT

EXPERIMENTS WRITING HIGHER-ORDER ABSTRACTIONS IN ELIXIR
OR: PUTTING THE "FUN" BACK IN "FUNCTOR"

FP is a set of principles and practice, rather than one monolithic thing.
We should embrace different ways of achieving these aims.

Different results of FP principles

- Elixir
  - Can feel (somewhat) imperative
    - Lots of operational logic
  - Thinks primarily in directional data “flow” (horizontal)

- Haskell
  - Largely declarative
  - Often think in abstractions (vertical)

Different results of FP principles

- Crossover
  - Haskell has **pipes**
  - Elixir has **Enum**
- I still want to try getting more “Haskell in Elixir” ¯_(ツ)_/¯

Adding a Vertical Dimension to Elixir

WITCHCRAFT

Witchcraft

1. **Witchcraft** (also called witchery or spellcraft) broadly means the practice of, and belief in**, magical skills and abilities** that are able to be exercised individually, by designated social groups, or by **persons with the necessary esoteric secret knowledge**

2. A category-inspired **library for Elixir**

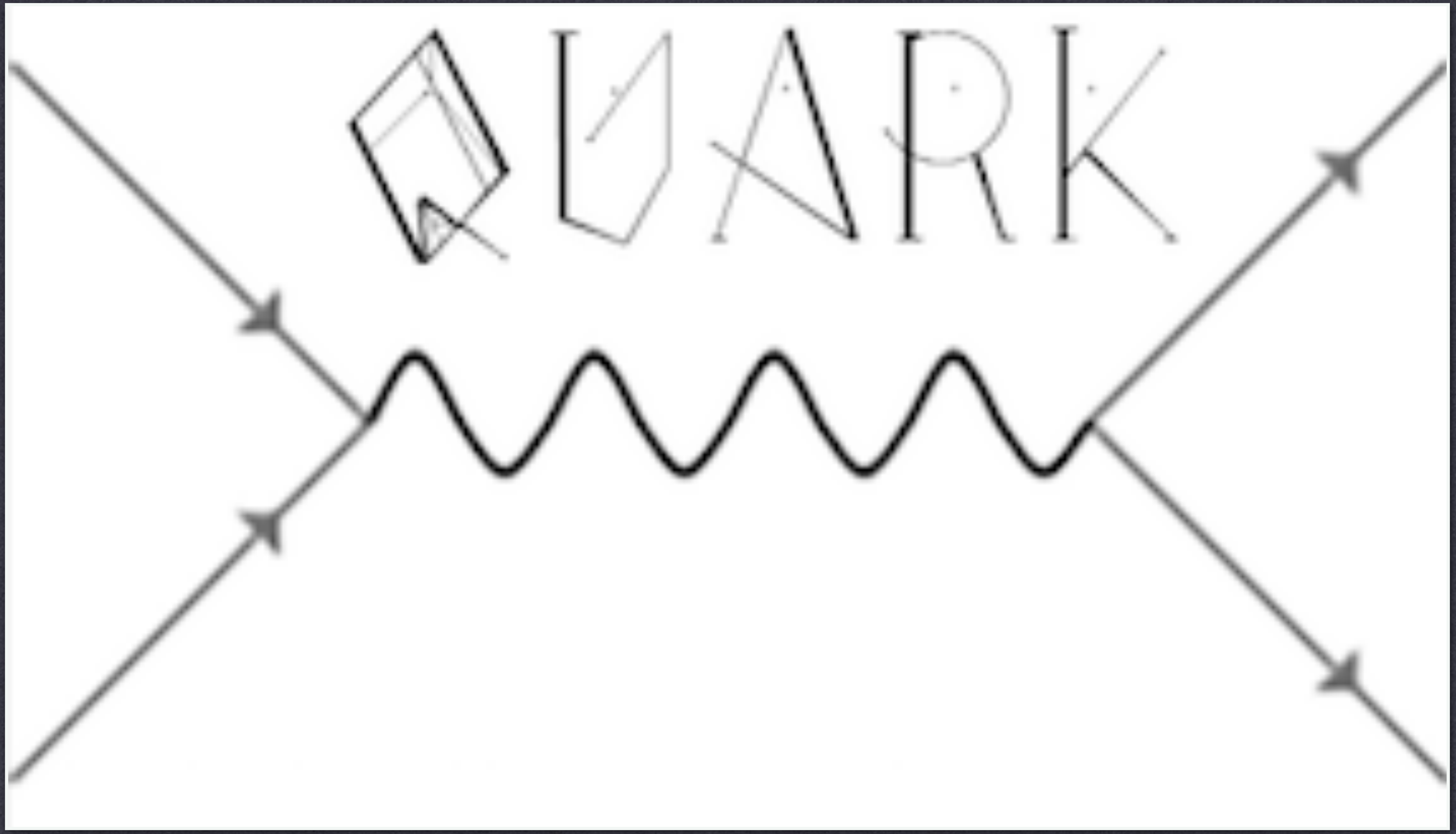
Witchcraft

- **Monoid**, **Functor**(s), **Monad**, **Arrow**, and **Category** for Elixir
- Follows the Haskell **Prelude** and **Control** modules pretty closely
- A lot of these rely on combinators and currying

Want partial application in Elixir

- Elixir is an arity-based language
  - (Automatic) partial application isn't a thing 😱
  - Currying isn't a thing
  - **foo(a)** is a different function from **foo(a, b)**
- Bootstrap time!
  - ***Massive detour…***

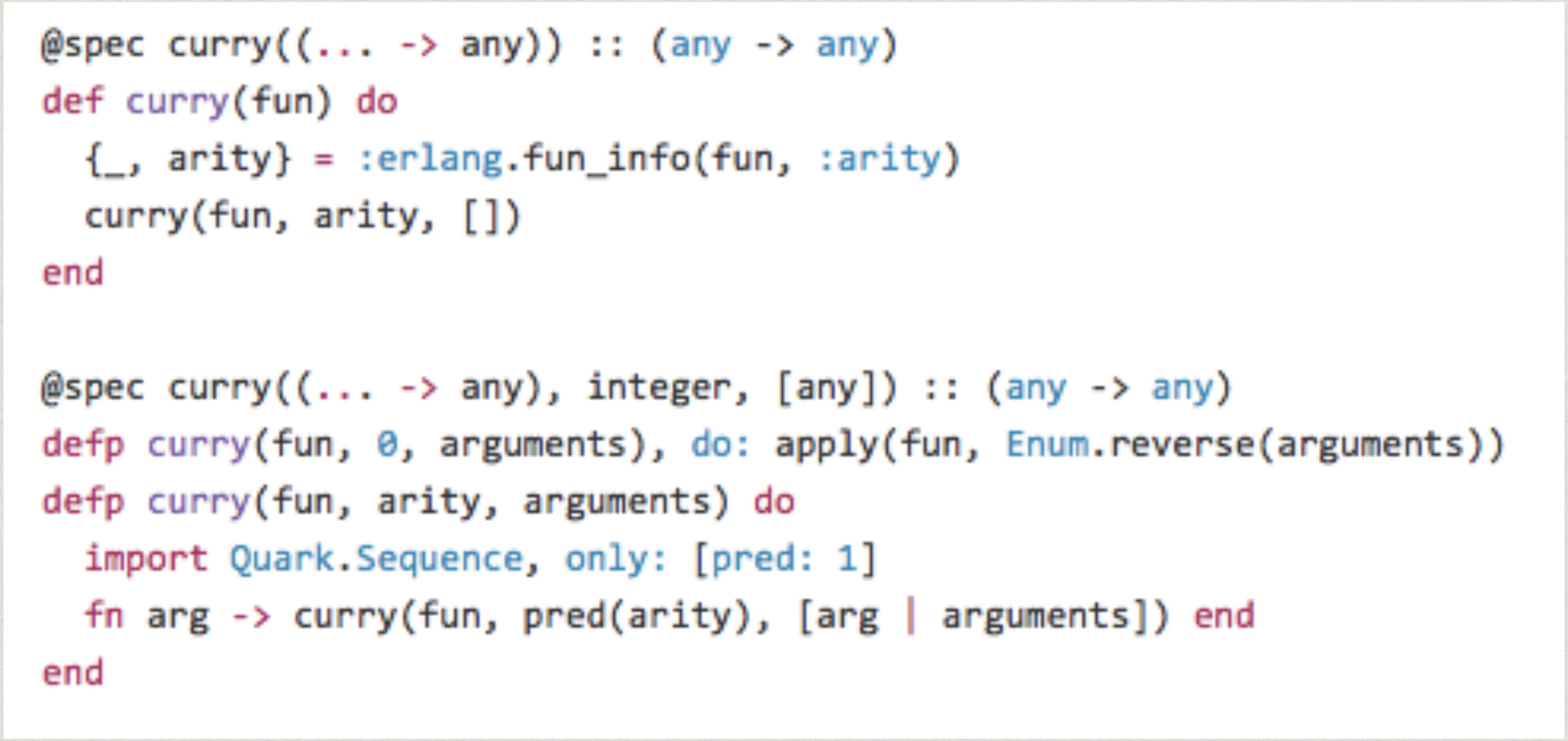
MASSIVE DETOUR



Quark

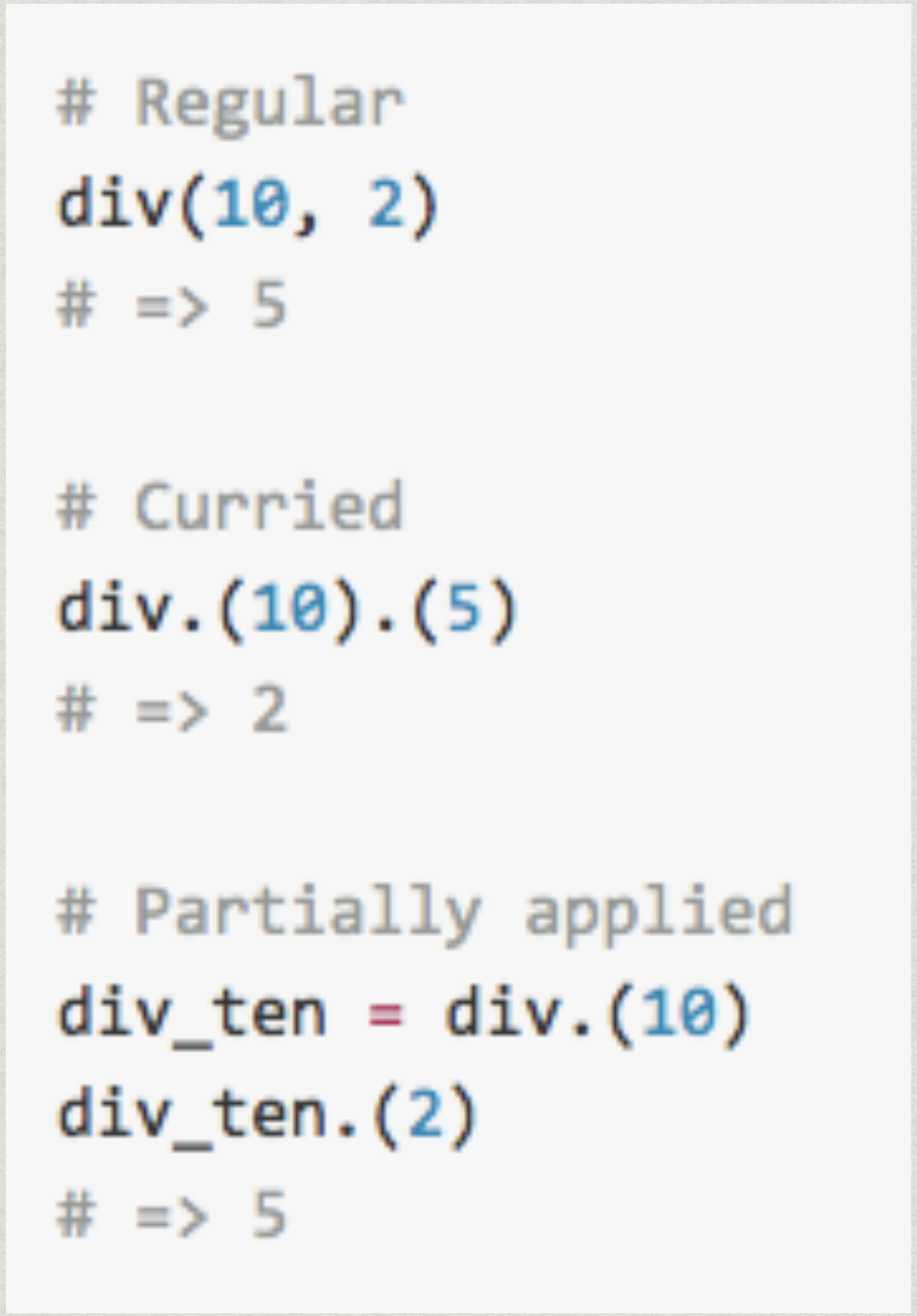
- Combinators for Elixir (**id**, **flip**, **const**, **fix**, **SKI**, &c)
  - How does Elixir now have these in the standard lib?!
- Currying and (completely faked) partial application

Runtime Currying in Elixir



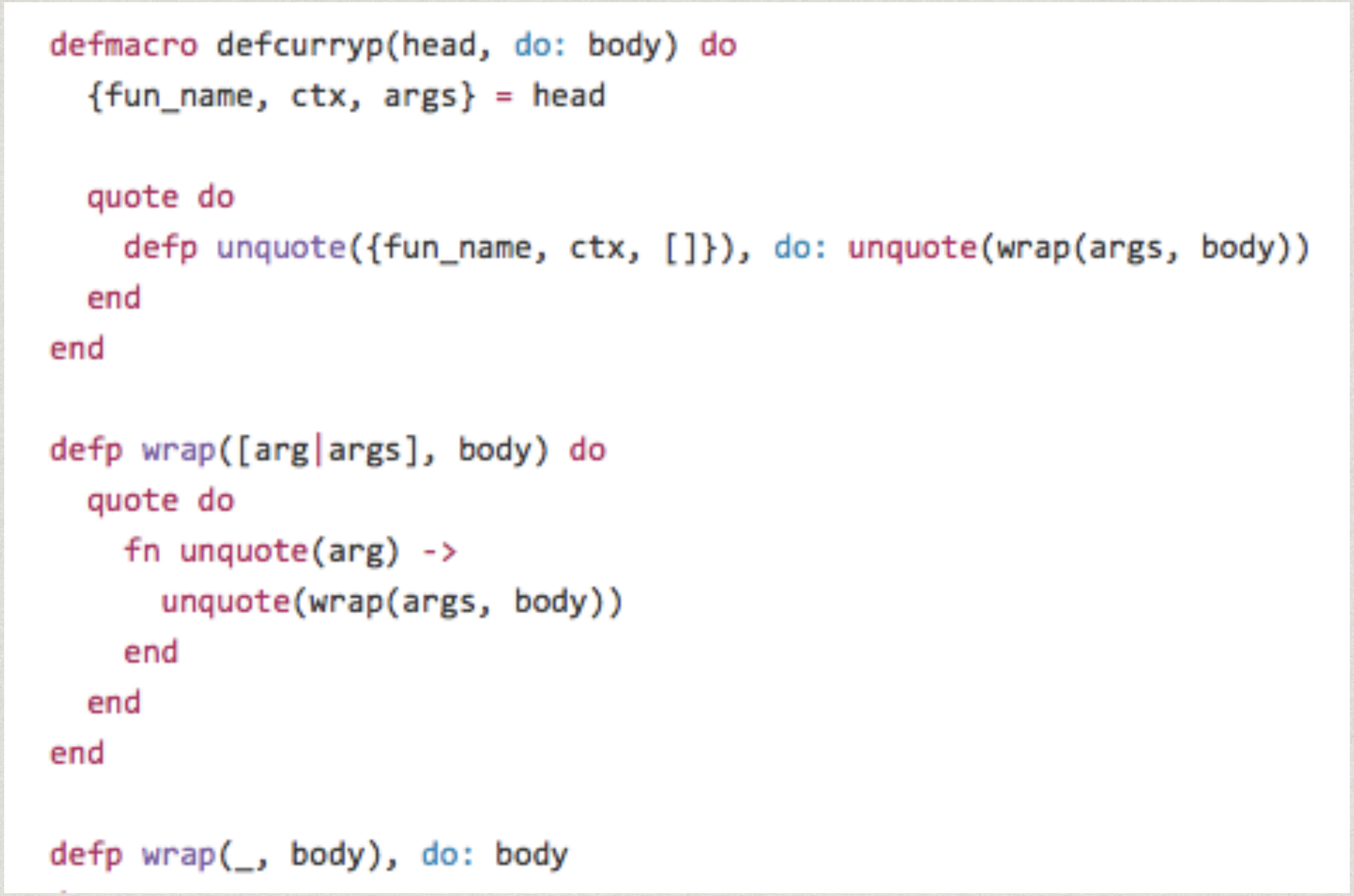
```elixir
@spec curry((... -> any)) :: (any -> any)
def curry(fun) do
  {_, arity} = :erlang.fun_info(fun, :arity)
  curry(fun, arity, [])
end

@spec curry((... -> any), integer, [any]) :: (any -> any)
defp curry(fun, 0, arguments), do: apply(fun, Enum.reverse(arguments))
defp curry(fun, arity, arguments) do
  import Quark.Sequence, only: [pred: 1]
  fn arg -> curry(fun, pred(arity), [arg | arguments]) end
end
```

Runtime Currying in Elixir



```
# Regular
div(10, 2)
# => 5

# Curried
div.(10).(5)
# => 2

# Partially applied
div_ten = div.(10)
div_ten.(2)
# => 5
```

Compile-Time Currying in Elixir



```elixir
defmacro defcurryp(head, do: body) do
  {fun_name, ctx, args} = head

  quote do
    defp unquote({fun_name, ctx, []}), do: unquote(wrap(args, body))
  end
end

defp wrap([arg|args], body) do
  quote do
    fn unquote(arg) ->
      unquote(wrap(args, body))
    end
  end
end

defp wrap(_, body), do: body
```

defpartial

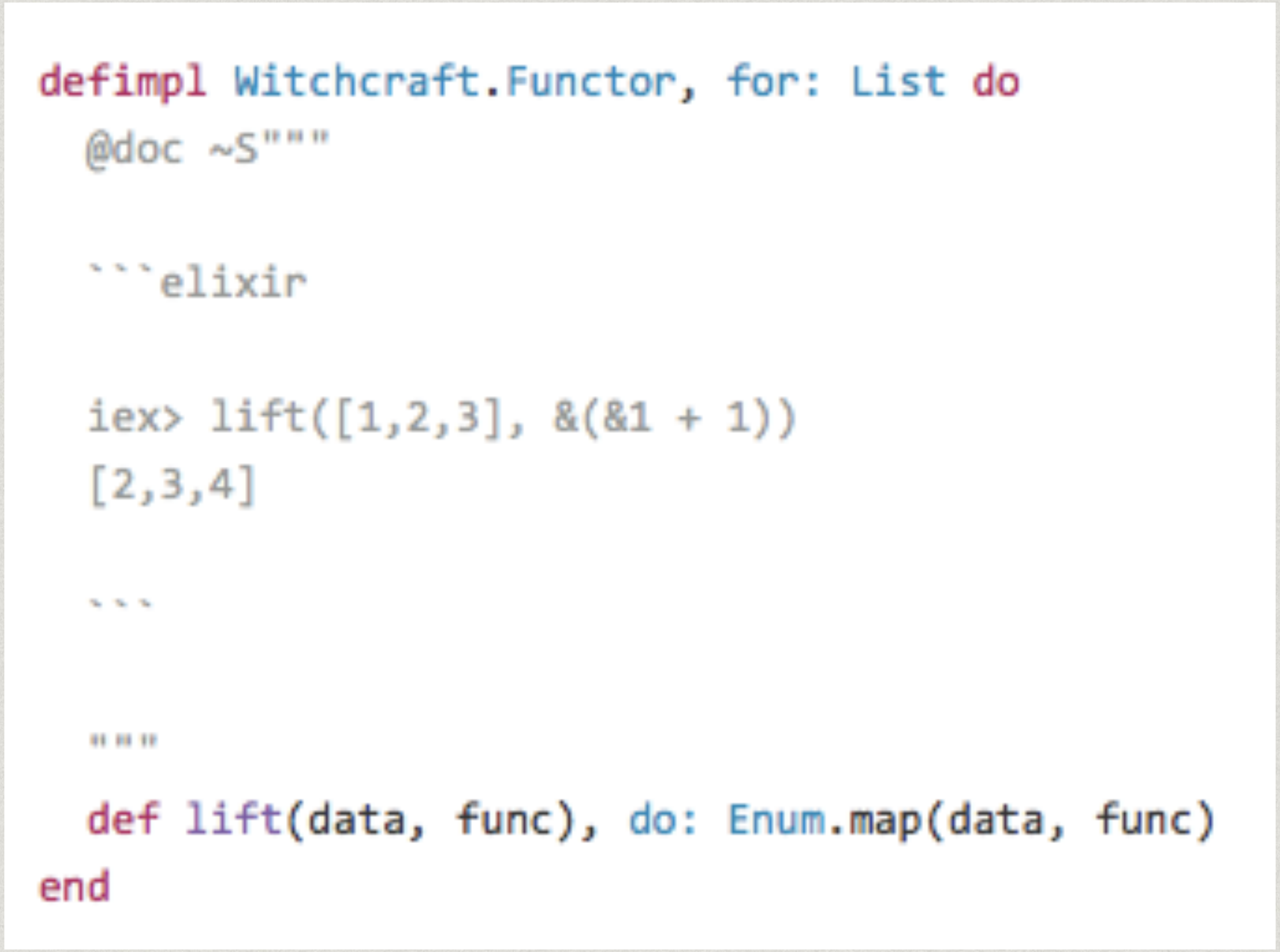
- Destroys the Elixir arity system 😅
- Still really nice to use *internally*
- Will get folded back in to **defcurry** eventually
  - Need to be able to specify **only** and **except**

BACK TO

Back to Witchcraft

- Functors, monads, arrows, categories for Elixir
- Follows the Haskell **Prelude** and **Control** modules pretty closely
- A lot of these rely on combinators and currying

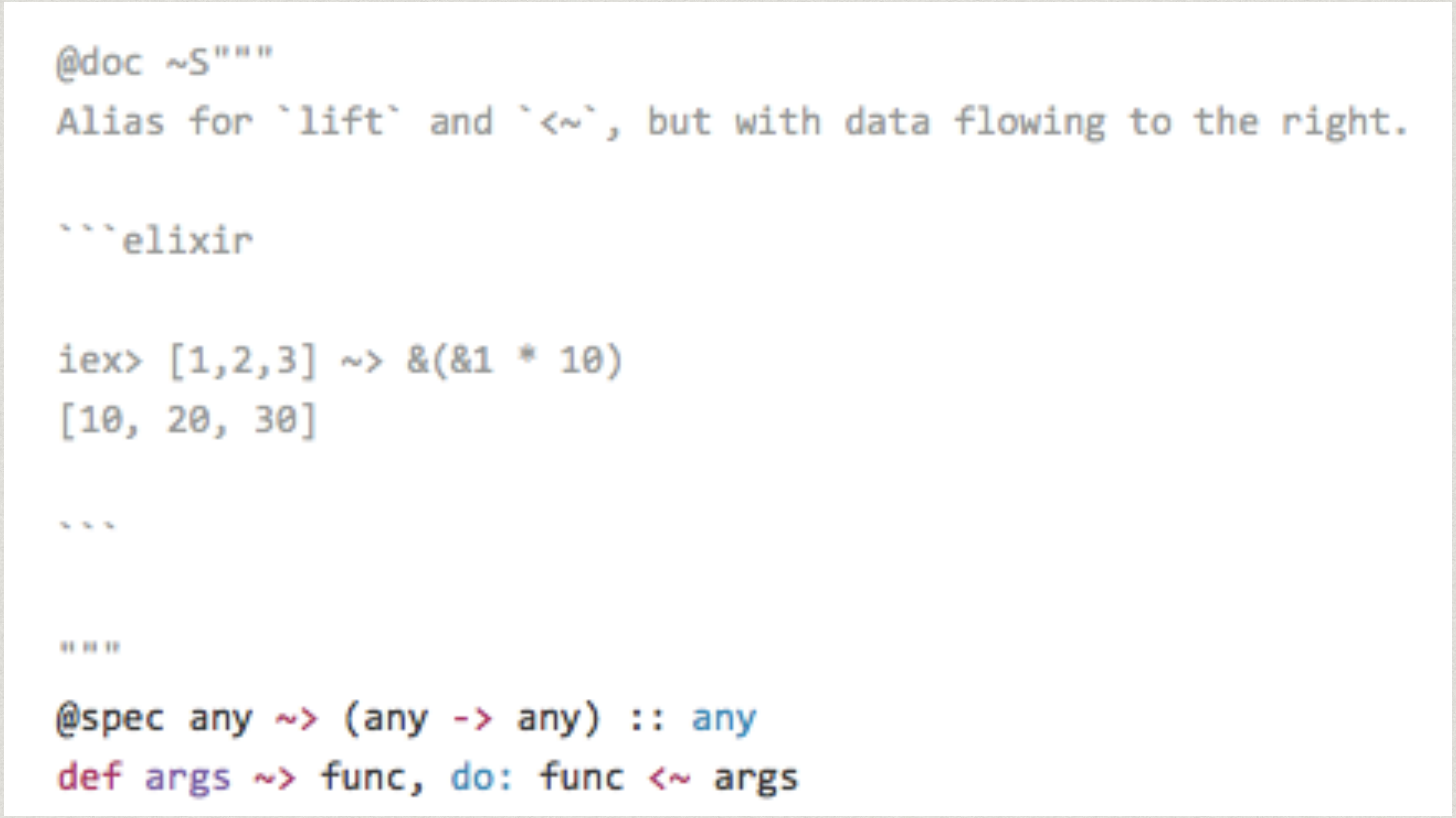
Just Protocols & Functions



~~~elixir
defimpl Witchcraft.Functor, for: List do
  @doc ~S"""

  ```elixir

  iex> lift([1,2,3], &(&1 + 1))
  [2,3,4]

  ~~~

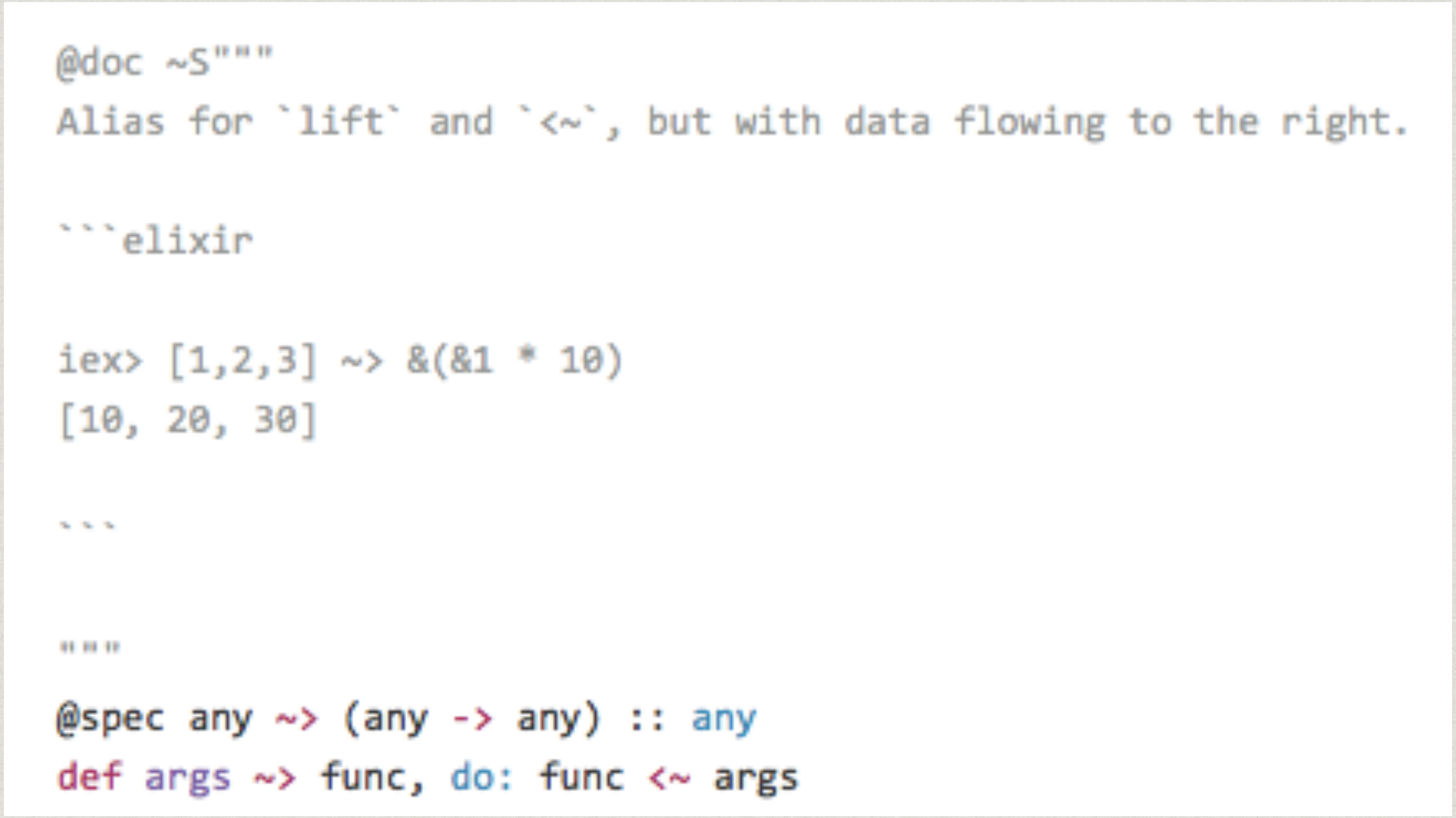
  """
  def lift(data, func), do: Enum.map(data, func)
end
```
```

Operators



~~~
@doc ~S"""
Alias for `lift` and `<~`, but with data flowing to the right.

```elixir

iex> [1,2,3] ~> &(&1 * 10)
[10, 20, 30]

```


"""
@spec any ~> (any -> any) :: any
def args ~> func, do: func <~ args
~~~

Operators are Backwards?!

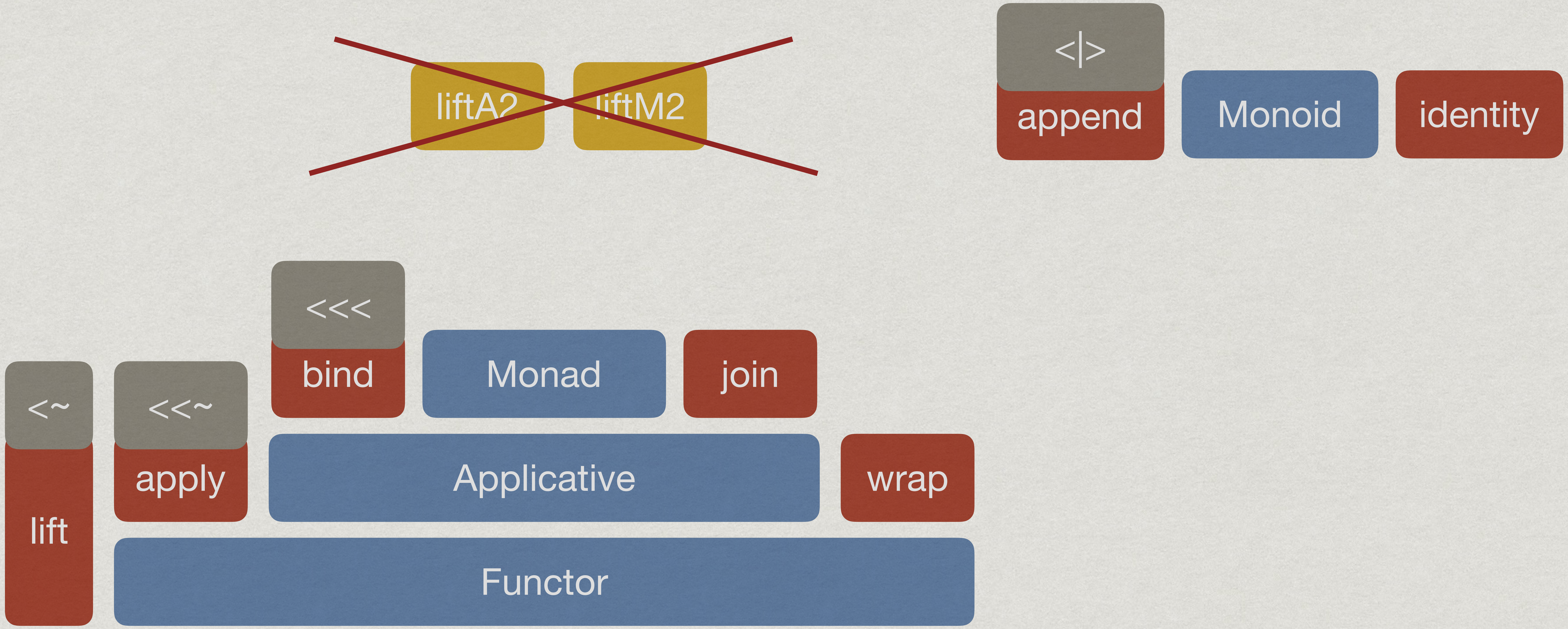


~~~
@doc ~S"""
Alias for `lift` and `<~`, but with data flowing to the right.

```elixir

iex> [1,2,3] ~> &(&1 * 10)
[10, 20, 30]

```


"""
@spec any ~> (any -> any) :: any
def args ~> func, do: func <~ args
~~~

The Operators are Backwards?!

- Philosophical difference in Elixir
    - Thinking horizontally (in “flow”) == data is the primary “subject”

Witchcraft so far



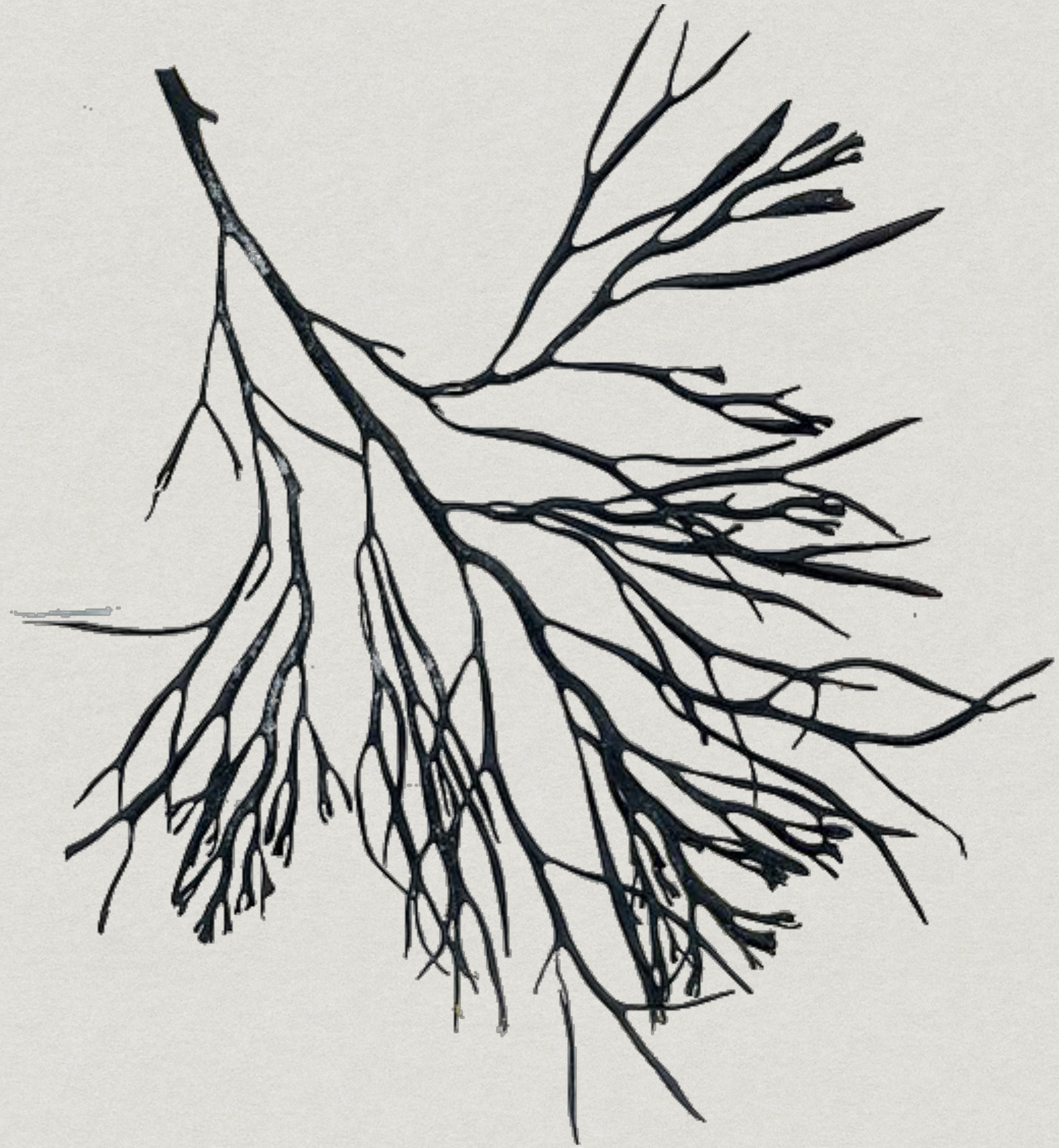
ADTs

- Want ADTs to get the most out of Witchcraft
- Elixir doesn’t have ADTs…

BUT ELIXIR HAS STRUCTS

Algae

Bootstrapped
algebraic data types
for Elixir



```elixir
defimpl Witchcraft.Applicative, for: Witchcraft.Id do
  import Quark.Curry, only: [curry: 1]
  alias Witchcraft.Id, as: Id
```

~~~elixir
@doc ~S"""

```elixir

iex> %Witchcraft.Id{} |> wrap(9)
%Witchcraft.Id{id: 9}

~~~

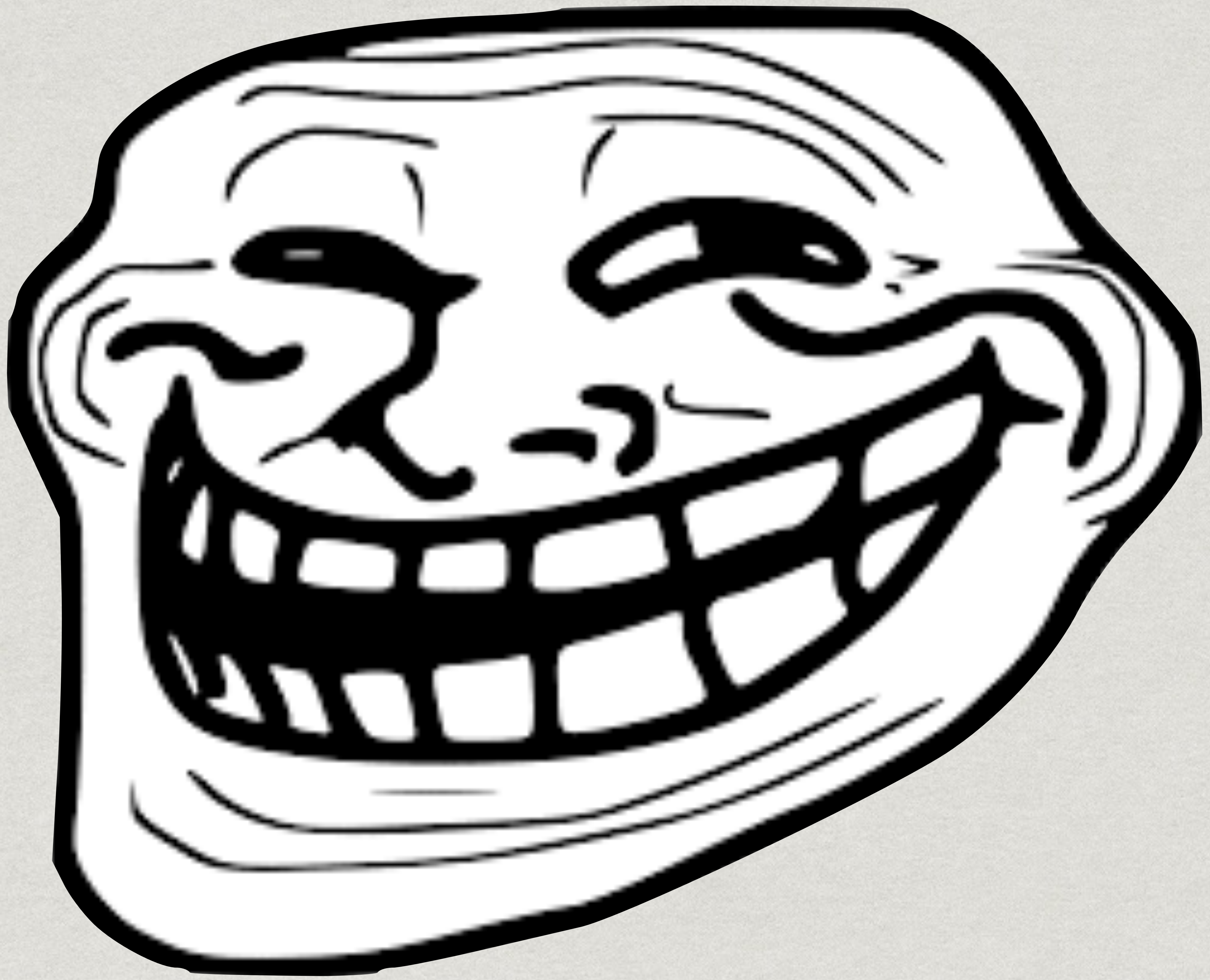
"""
def wrap(_, bare), do: %Witchcraft.Id{id: bare}
~~~

```elixir
@doc ~S"""
```elixir

iex> import Kernel, except: [apply: 2]
iex> apply(%Witchcraft.Id{id: 42}, %Witchcraft.Id{id: &(&1 + 1)})
%Witchcraft.Id{id: 43}

iex> import Kernel, except: [apply: 2]
iex> import Witchcraft.Functor, only: [lift: 2]
iex> alias Witchcraft.Id, as: Id
iex> apply(%Id{id: 9}, lift(%Id{id: 2}, &(fn x -> x + &1 end)))
%Witchcraft.Id{id: 11}

~~~

"""
def apply(%Id{id: value}, %Id{id: fun}), do: %Id{id: curry(fun).(value)}

```elixir
end
```

Algae

- Internals are the topic of another talk



Q&A

Fin