MONADS FOR THE WORKING ALCHEMIST

WITCHCRAFT

LOTS OF STUFF

~~MONADS~~ FOR THE WORKING ALCHEMIST

WITCHCRAFT

ABOUT THIS TALK

- The Witchcraft suite: easier algebraic code for Elixir
- Don’t need to know Haskell/Idris/PureScript/cat theory, but it helps
- Throwing a lot at you (sorry)
- Only covering a subset of what’s in these libraries
- One takeaway:

BROOKLYN ZELENKA

- @expede
- Founding partner at Robot Overlord
- Runs the Vancouver Functional Programming and Vancouver Erlang/Elixir Meetups

- Author of Witchcraft, TypeClass, Algae, Quark, Exceptional, and others
- I have Witchcraft stickers (come see me afterwards)

WHAT'S IN A NAME?

witchcraft noun /ˈwɪtʃˌkrɑːft/

1. the art or power of bringing magical or preternatural power to bear or the act or practice of attempting to do so

2. the influence of magic or sorcery

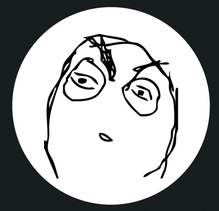
3. fascinating or bewitching influence or charm

4. an Elixir library

WHAT'S SO MYSTICAL ABOUT IT?

- High abstraction is powerful but can look intimidating to newcomers
- "Monads"
- Category theoretic basis
  - Don't *need* to know it
  - Hask → Ex

```
&(&1 + &2 * &3) <~ [1, 2, 3] <<~ [4, 5, 6] <<~ [7, 8, 9]
[
  29, 33, 37,
  36, 41, 46,
  43, 49, 55,
  30, 34, 38,
  37, 42, 47,
  44, 50, 56,
  31, 35, 39,
  38, 43, 48,
  45, 51, 57
]
```



WAT

WHY ALL THE EXTRA LIBRARIES?

MAKING THE CASE

ALGEBRAIC CODE

- “Algebra” here just means “a set of rules for manipulating related objects”
- You can make your own algebras
  - Indeed you do without thinking about it
  - For example, many DSLs and protocols

ALGEBRAIC CODE

- High confidence
- Reusable abstractions (write once, run in many contexts)
- Declarative, pure/concurrent safe code
- More tools in your toolbox
- Convenience
- Well understood, principled, *functional design patterns*
  - Lenses, free monad + interpreter, DSLs
- Unify similar concepts through abstraction
  - `+/2`, `++/2`, `<>/2`, `Map.merge/2`, `MapSet.union/2` have a *lot* in common (semigroup)

PRINCIPLES DRIVING

DESIGN

Compatibility with Elixir ecosystem

Consistency with mental models

Portability from other ecosystems

Pedagogy and approachability

I would like to add a slightly different perspective to functional programming in the Erlang VM: functional programming is not a goal in the Erlang VM. It is a means to an end.

When designing the Erlang language and the Erlang VM, Joe, Mike and Robert did not aim to implement a functional programming language, they wanted a runtime where they could build distributed, fault-tolerant applications. It just happened that the foundation for writing such systems share many of the functional programming principles. And it reflects in both Erlang and Elixir.

–JOSE VALIM

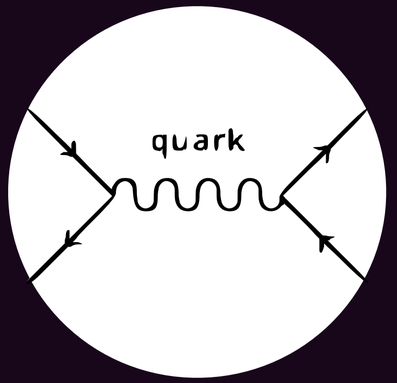
http://blog.plataformatec.com.br/2016/05/beyond-functional-programming-with-elixir-and-erlang/

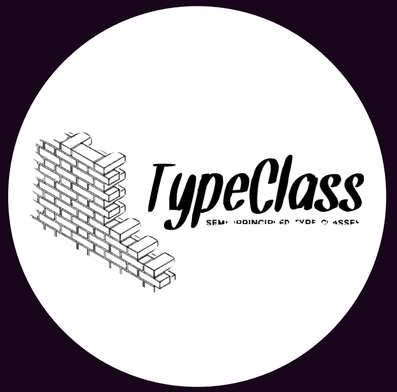
How can I make this more like Haskell? (Bulb paradox)

–BROOKLYN ZELENKA

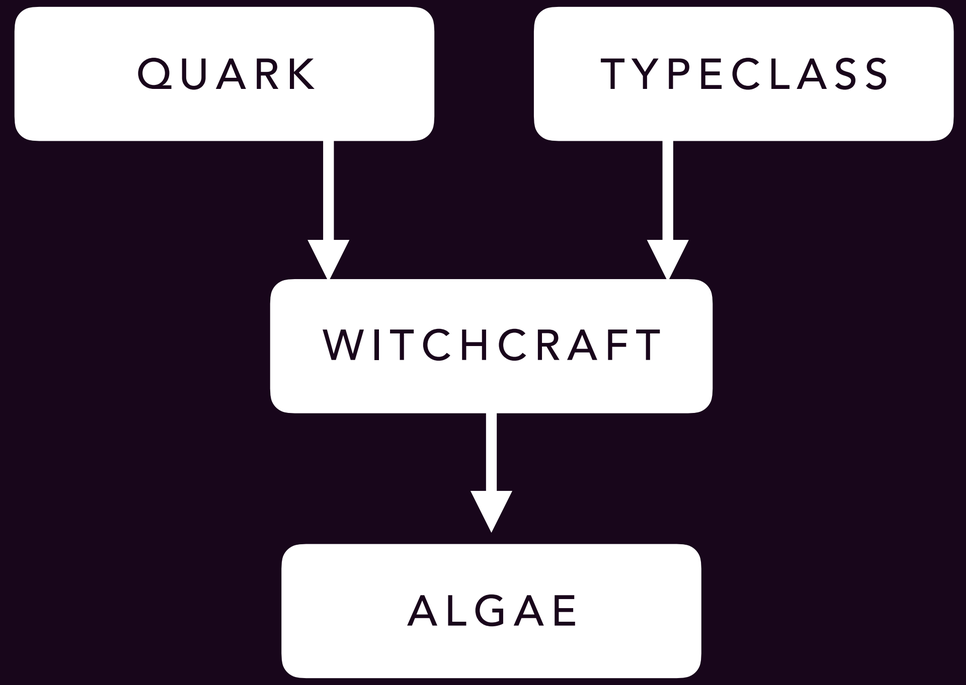
QUARK • TYPECLASS • ALGAE

BOOTSTRAPPING



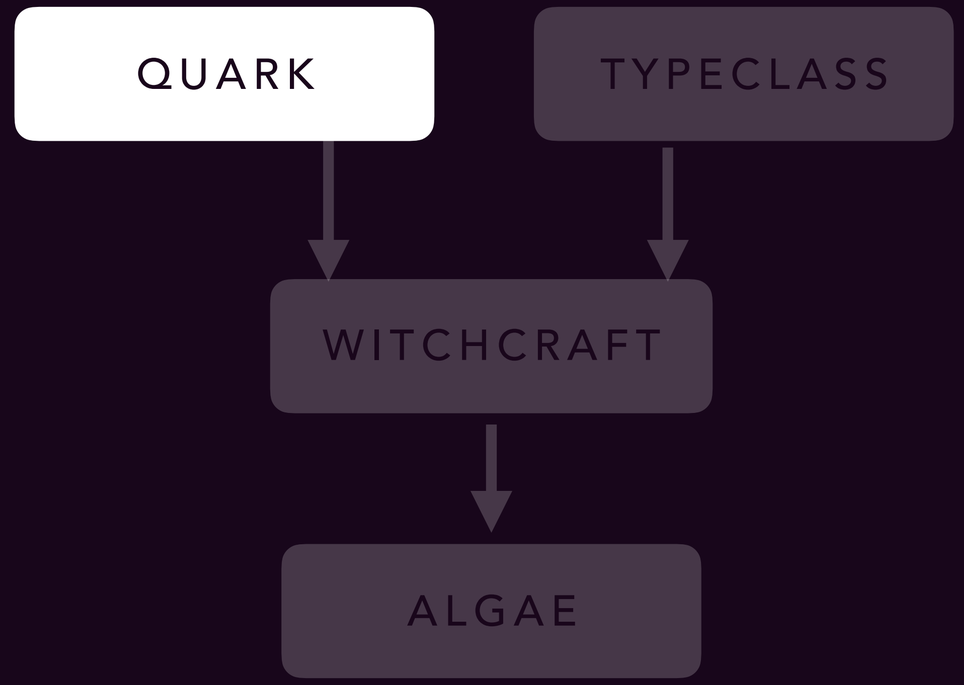


BOOTSTRAPPING



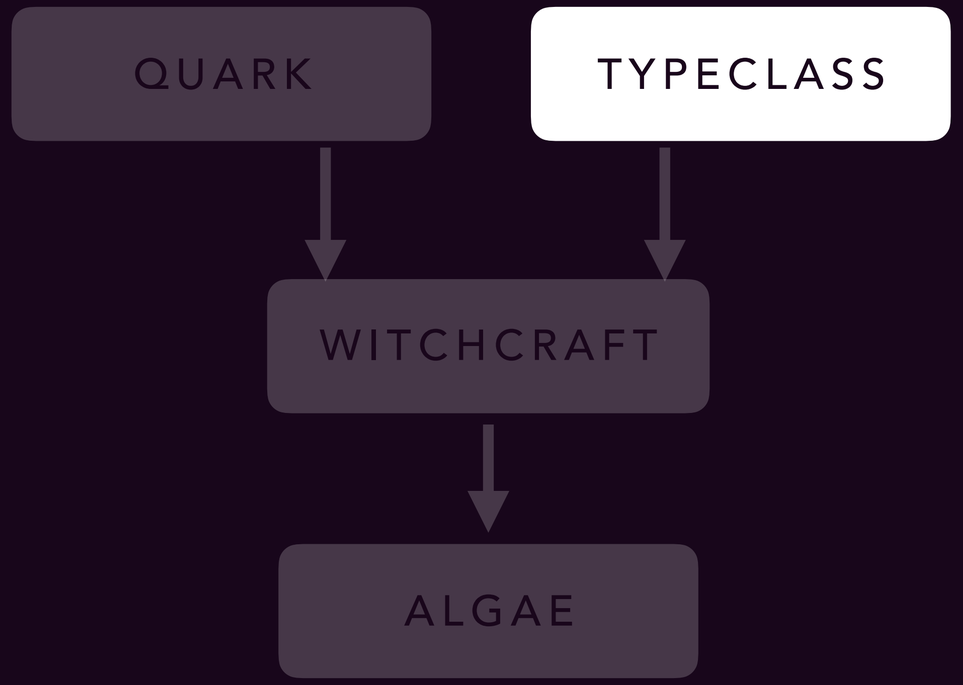
QUARK

- Classic combinators
  - `id`, `flip`, `const`, and so on
- Function composition
- Currying
  - Very important for Witchcraft!
  - Some constructs must be curried,
    and for convenience we do this dynamically



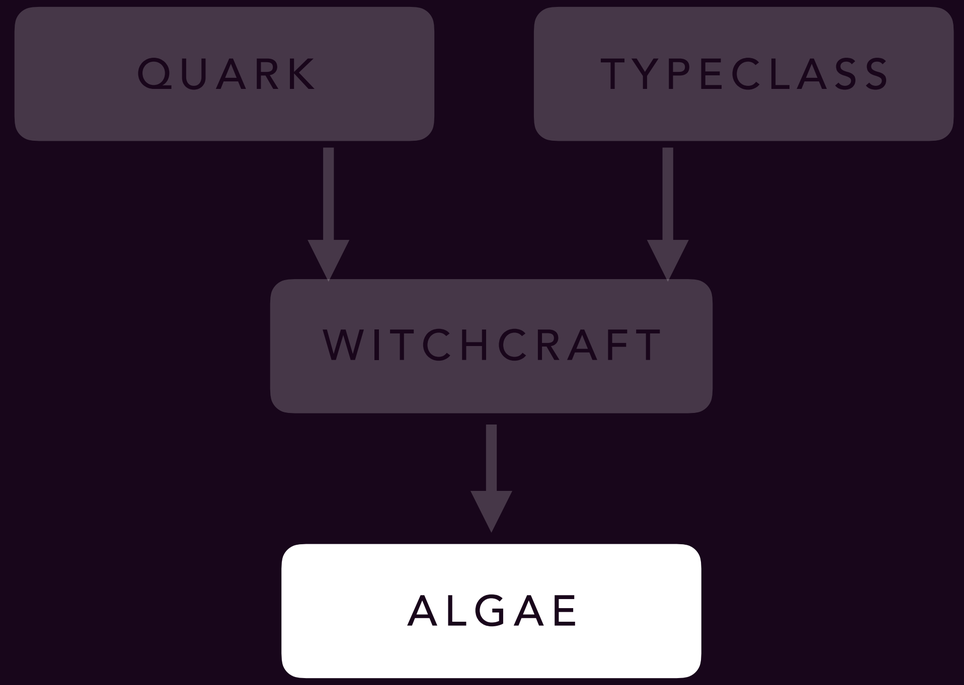
TYPECLASS

- Principled type classes
- “Classes” and “methods” not the same as in OO
- Properties & methods
- Hierarchies
- Hiding side protocols from API
  - Enum + Enumerable



ALGAE

- DSL for writing related structs



ALGAE

`defdata`

All of these fields

Roughly “and”

```
defdata do
  name        :: String.t()
  hit_points  :: non_neg_integer()
  experience  :: non_neg_integer()
end
```

`defsum`

One of these structs

Roughly “or”

```
defsum do
  defdata Nothing :: none()
  defdata Just    :: any()
end
```

ALGAE

```elixir
defsum do
  defdata Nothing :: none()
  defdata Just    :: any()
end
```

```elixir
defmodule Algae.Maybe.Just do
  @type t :: %Algae.Maybe.Just{just: any()}
  defstruct [just: nil]

  def new, do: %Algae.Maybe.Just{}
  def new(value), do: %Algae.Maybe.Just{just: value}
end

defmodule Algae.Maybe.Nothing do
  @type t :: %Algae.Maybe.Nothing{}
  defstruct []

  def new, do: %Algae.Maybe.Nothing{}
end
```

ALGAE.MAYBE

- Many uses, but can be “something or failure”

```elixir
alias Algae.Maybe.{Just, Nothing}

def add(x, y) do
  try do
    Just.new(x + y)
  rescue
    _ -> %Nothing{}
  end
end
```

```
iex> add(1, 2)
%Just{just: 3}

iex> add(1, "NOPE")
%Nothing{}
```

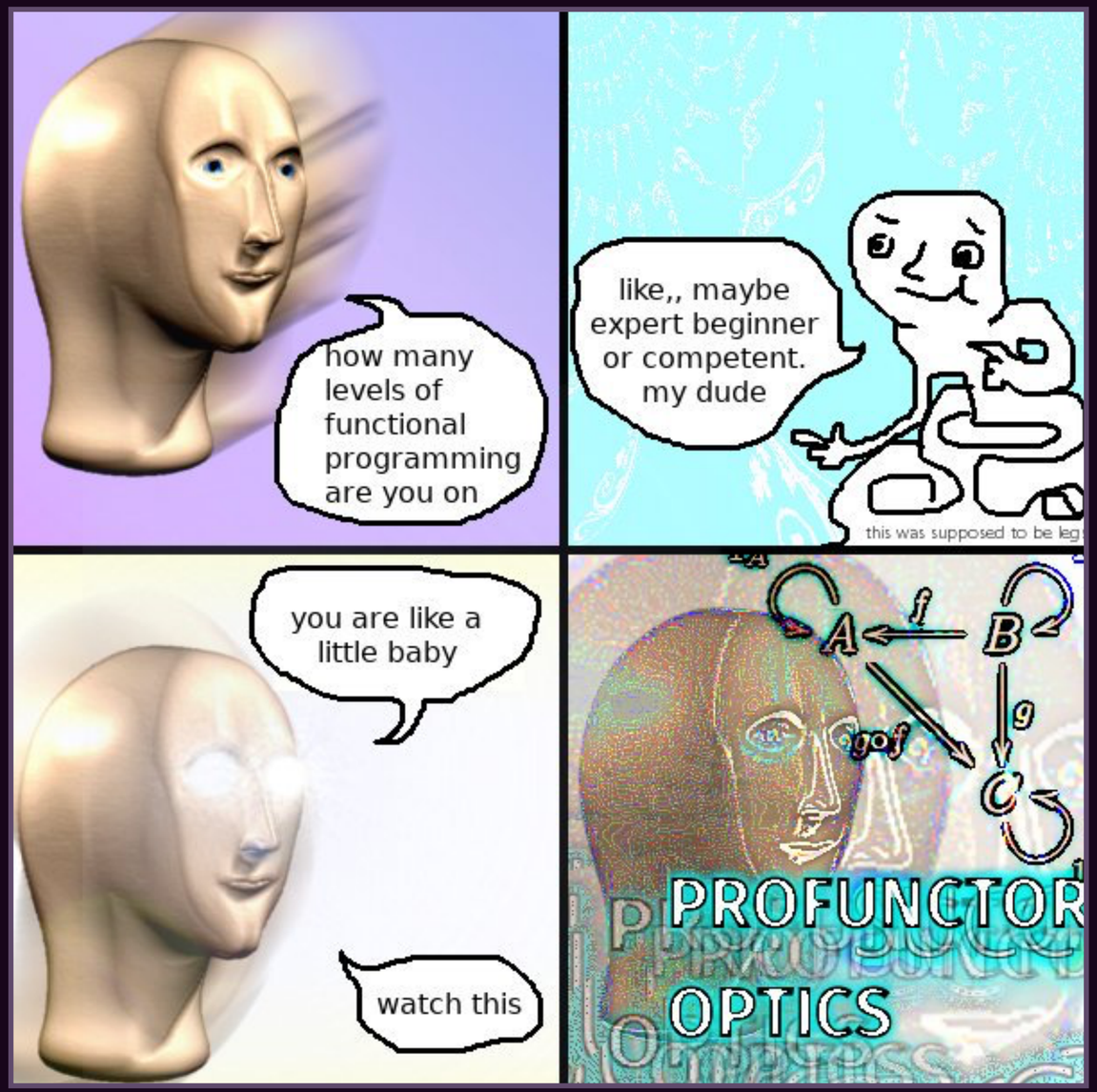
KEEPING IT ELIXIR-Y

CONSISTENCY & ETHOS

CONSISTENCY & ETHOS

- Introducing these concepts in other languages has been uneven
  - Scalaz and Swiftz famously called “Haskell fan fiction”
    - Some Scala looks like Java, some looks like Haskell, and hard to intermingle the two
- Elixir
  - Borrowed the friendly community from Ruby
  - Data flow and function application over classical expressions and composition
  - Dynamically typed, or “type checked at runtime”

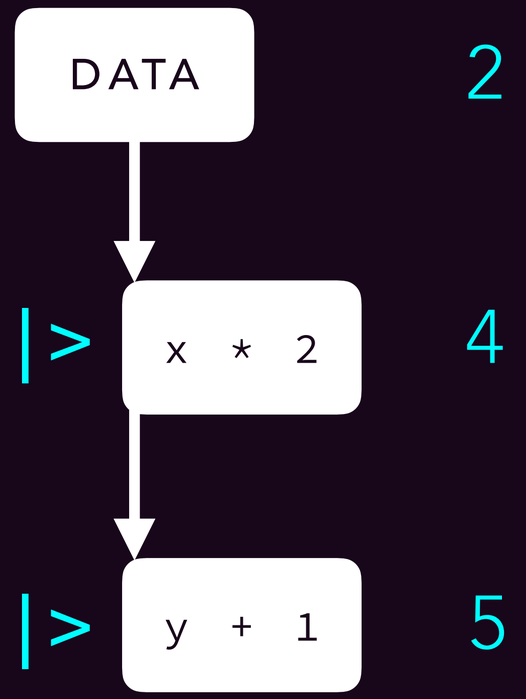
CONSISTENCY & ETHOS



The feeling that we're trying to avoid

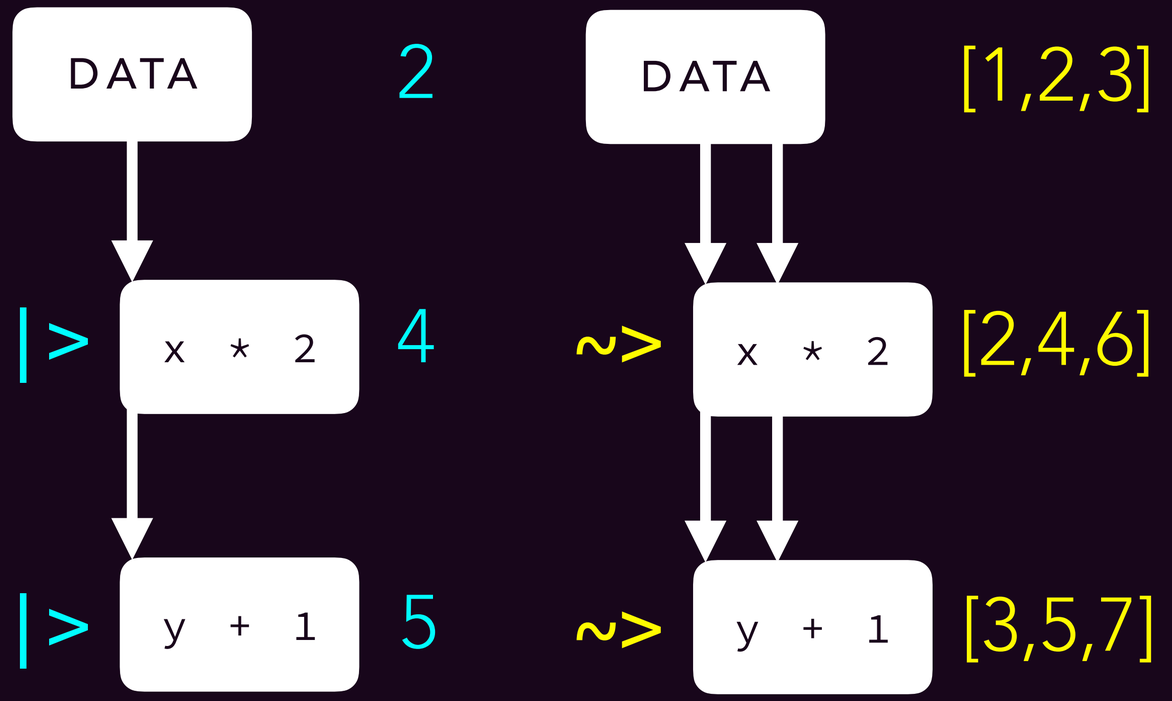
DIRECTIONALITY & DATAFLOW

- Let’s bootstrap people’s intuitions!
- Elixir prefers diagrammatic ordering
- Important to maintain consistency with rest of language!
- Pipes are generally awesome
- Want to maintain this awesomeness
- What if we just gave the pipe operator superpowers?

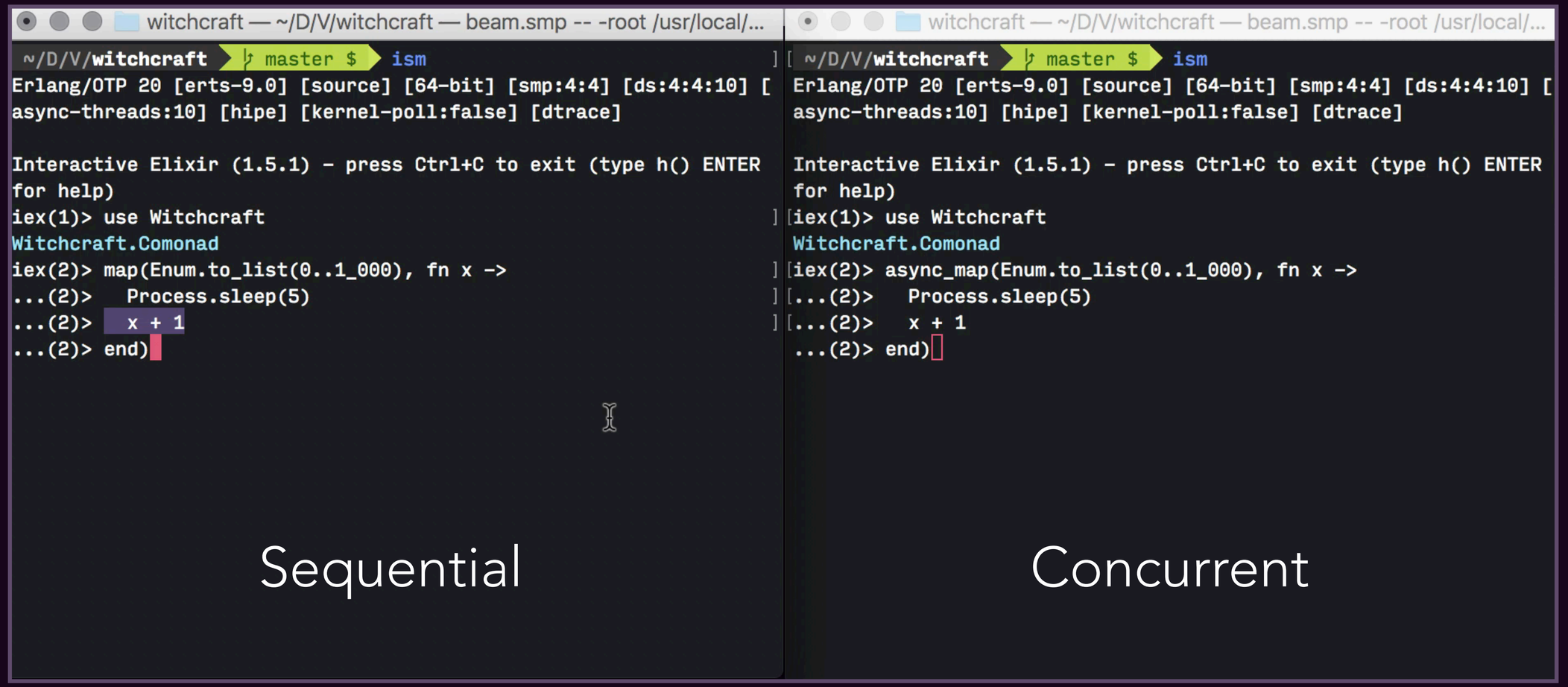


GIVING PIPES SUPERPOWERS 💪

- Witchcraft operators follow same flow
- Data on flows through pointed direction
  - Just like pipes
  - `|>` becomes **`~>`** (curried `map/2`)



ASYNC VARIANTS



Sequential

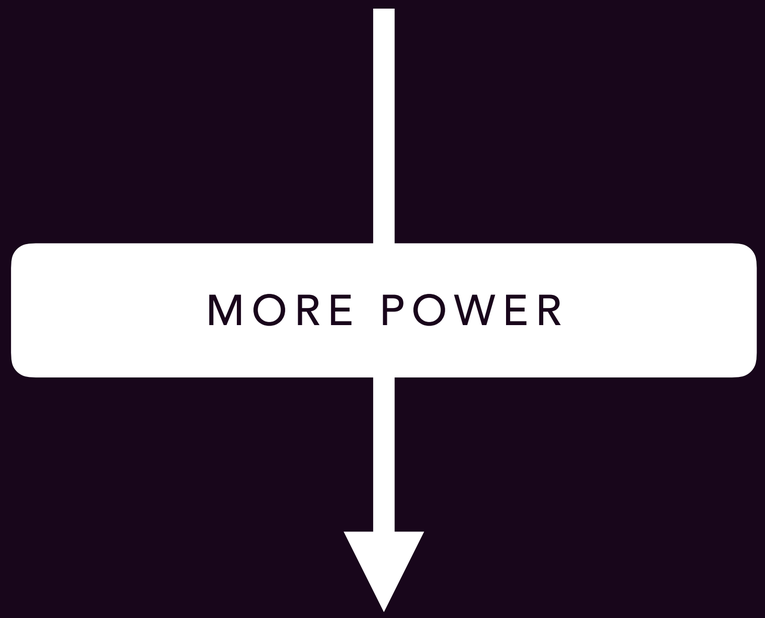
Concurrent

GIVING PIPES SUPERPOWERS 💪

- Operators follow same flow
- Data on flows through arrow direction

```
        [1, 2, 3] <~ fn x -> x * x end
fn x -> x * x end ~> [1, 2, 3]
```

- |>
- ~>
- ~>>
- >>>



(_) apply/2

<~ map/2

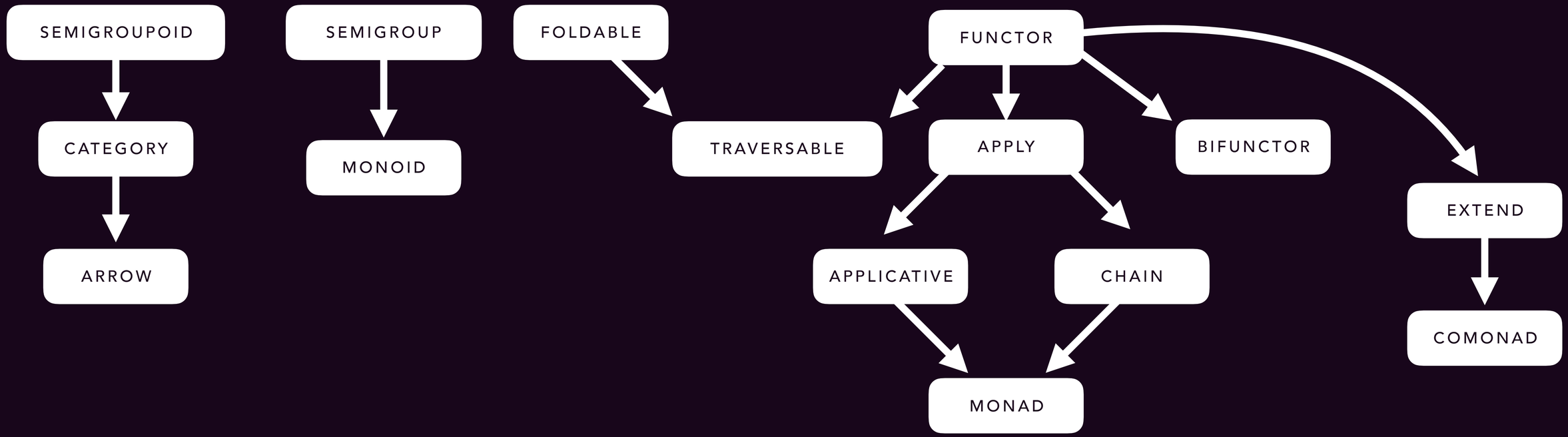
<<~ ap/2

<<< chain/2

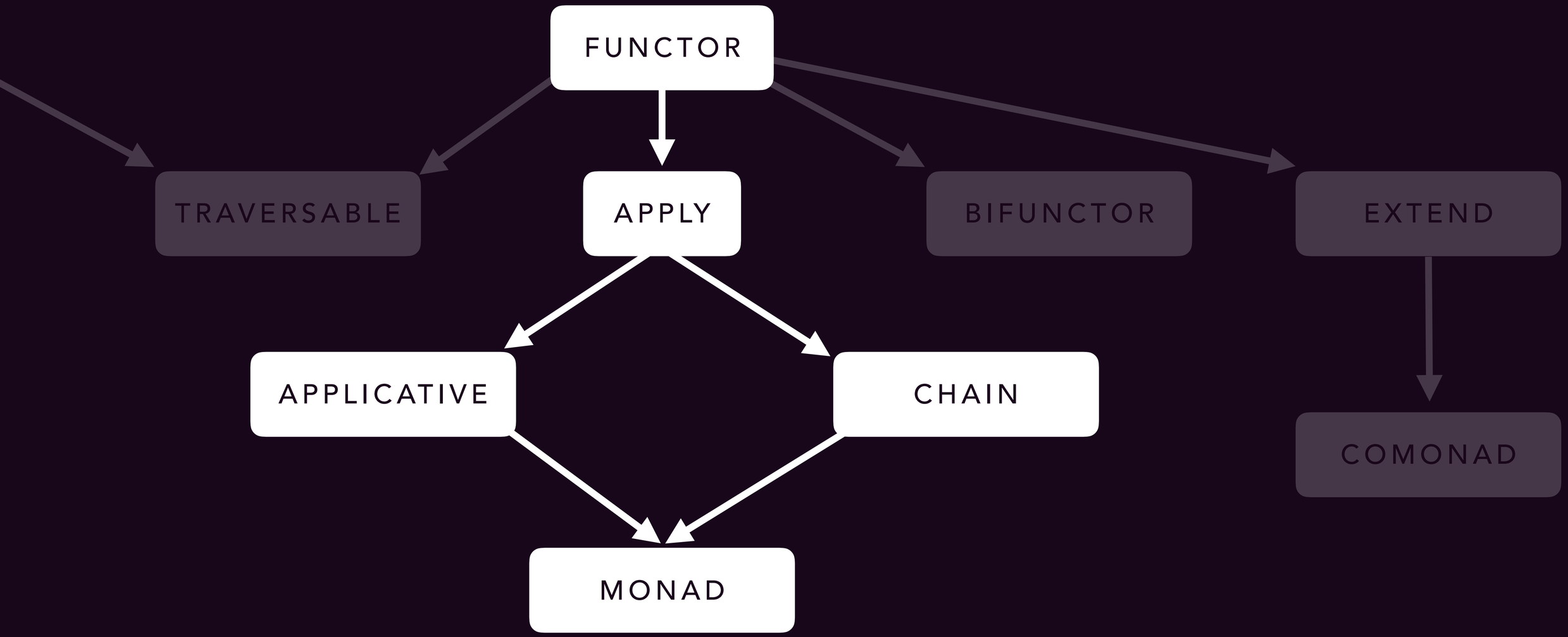
THESE ARE FUNCTIONAL & PRINCIPLED

DESIGN PATTERNS

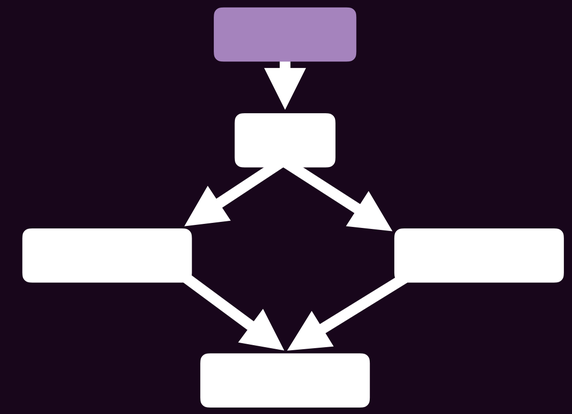
WITCHCRAFT 1.0 HIERARCHY



FUNCTOR HIERARCHY



FUNCTOR

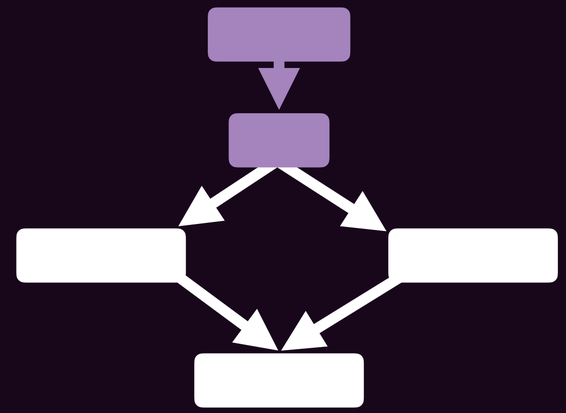


- Provides `map/2` (`~>`), but different from `Enum`
- Always returns the same type of data
  - No more manual `Enum.map(…) |> Enum.into(…)`

```elixir
Functor.map(%{a: 1, b: 2}, fn x -> x * 10 end)
#=> %{a: 10, b: 20}
```

```elixir
Functor.map(%Algae.Maybe.Just{just: 1}, fn x -> x * 10 end)
# => %Algae.Maybe.Just{just: 10}

Functor.map(%Algae.Maybe.Nothing{}, fn x -> x * 10 end)
#=> %Algae.Maybe.Nothing{}
```

APPLY: THINKING *INSIDE* THE BOX



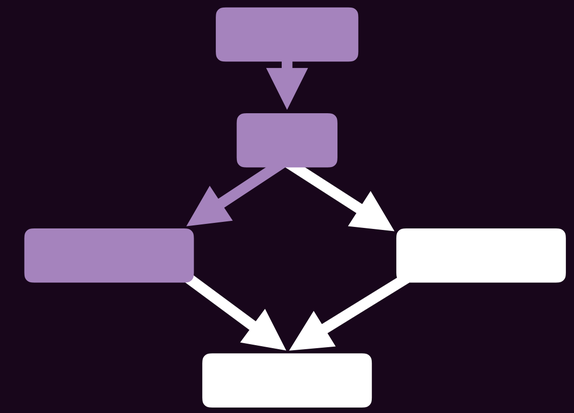
- Provides `convey/2` and `ap/2`
- Embellishes basic function application
- Specific embellishment changes per data type

```
iex> [1, 2, 3] ~>> [&(&1 * 10), &(&1 + 1), &(&1 - 5)]
[10, 2, -4, 20, 3, -3, 30, 4, -2]
```

```
iex> lift([1, 2, 3], [4, 5, 6], &*/2)
[
  4, 8,  12,
  5, 10, 15,
  6, 12, 18
]
```

```
iex> %Just{just: 1} ~>> %Just{just: fn x -> x * 10 end}
%Just{just: 10}

iex> %Nothing{} ~>> %Just{just: fn x -> x * 10 end}
%Nothing{}

iex> %Just{just: 1} ~>> %Nothing{}
%Nothing{}
```

APPLICATIVE: PUT INTO CONTEXT

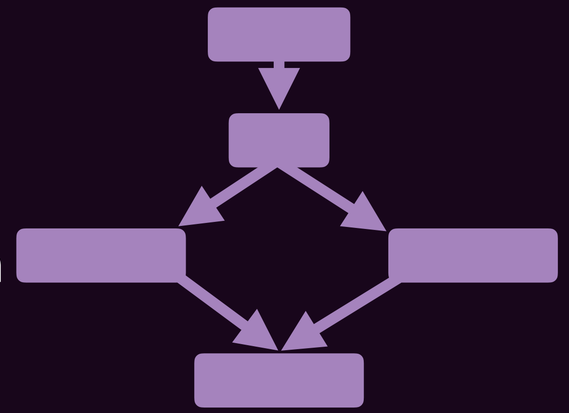


- Provide `of/2`
- Simple: lift values into a datatype
- Like `List.wrap/1`

```
iex> of([], 42)
[42]
```

```
iex> of(%Just{}, 42)
%Just{just: 42}
```

CHAIN: FUNCTIONS TO ACTIONS



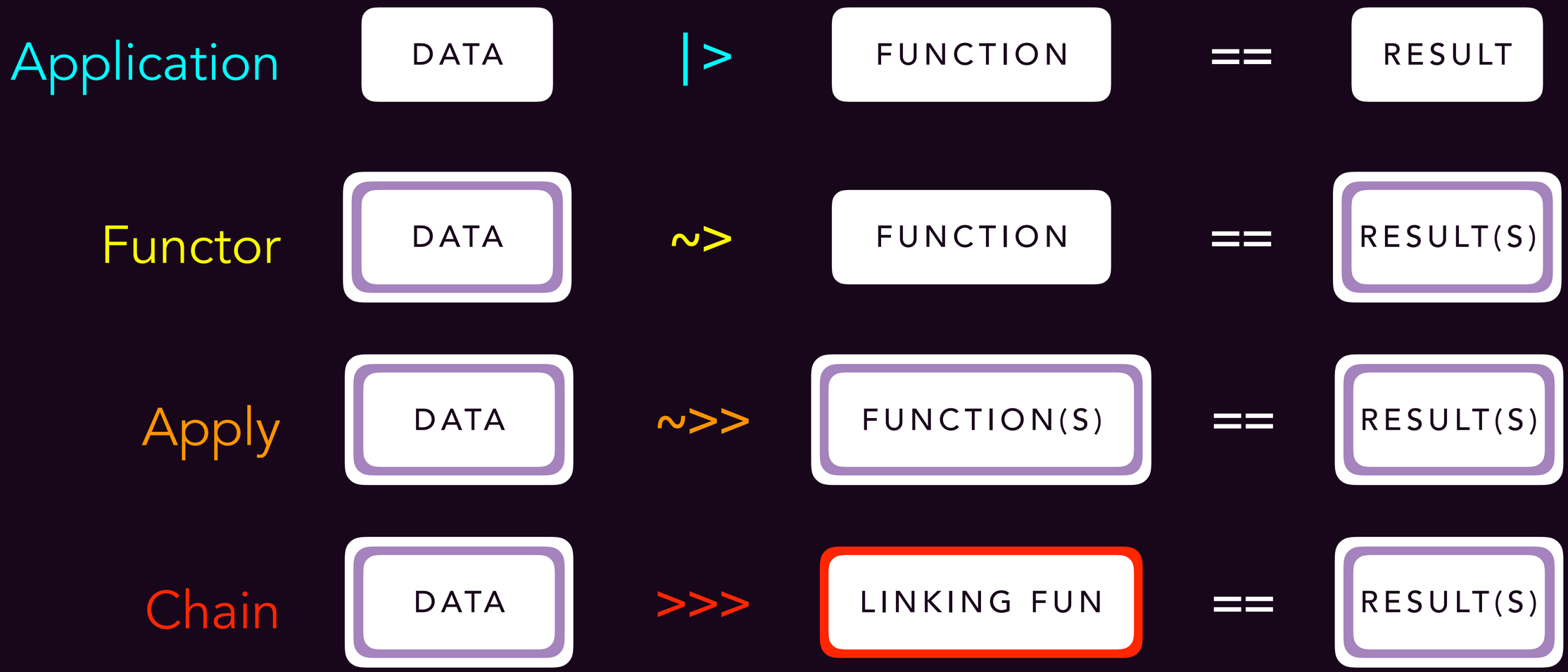
- Like **`Apply & Applicative`**, but with a special “linking” function
  - Take raw value
  - Do something to it
  - Put the result into the original datatype
  - Makes it easy to *chain* functions in a context

```
iex> [1, 2, 3] >>> fn x -> [x, x] end
[1, 1, 2, 2, 3, 3]
```

```
iex> [1, 2, 3]
...> >>> fn x -> [x, x] end
...> >>> fn y -> [y, y * 10, y * 100] end
[
  1, 10, 100,
  1, 10, 100,
  2, 20, 200,
  2, 20, 200,
  3, 30, 300,
  3, 30, 300
]
```

```
iex> %Just{just: 1}
...> >>> fn x -> if Integer.is_even(x), do: %Just{just: 42}, else: %Nothing{} end
...> >>> fn y -> if y > 10, do: %Just{just: 10}, else: %Just{just: y} end
...> >>> fn z -> %Just{just: z * z} end
%Nothing{} # 1 is not even, but was guarded
```

POWERING UP GRAPH



CHAIN DO NOTATION

- Macro to “linearize” chains
- Gives us back an operational feel
- Great DSLs (seen shortly)

```elixir
def guarded(input) do
  %Just{just: input}
  >>> fn x ->
    if Integer.is_even(x), do: %Just{just: x}, else: %Nothing{}
    >>> fn y ->
      if y > 0, do: %Just{just: 10}, else: %Just{just: y}
      >>> fn z ->
        %Just{just: x * y + z} # Access earlier steps
      end
    end
  end
end

guarded(1)   == %Nothing{}
guarded(100) == %Just{just: 1010}
```

```elixir
def do_guarded(input) do
  chain do
    x <- %Just{just: input}
    y <- if Integer.is_even(x), do: %Just{just: x}, else: %Nothing{}
    z <- if y > 0, do: %Just{just: 10}, else: %Just{just: y}
    Just{just: x * y + z}
  end
end

do_guarded(1)   == %Nothing{}
do_guarded(100) == %Just{just: 1010}
```

MONADIC DO NOTATION

- Need to specify the data type
- Just add `return` (specialized `Applicative.of/2`)

```elixir
def madlib(nouns, adjectives, verbs, reactions) do
  monad [] do
    noun  <- nouns
    adj   <- adjectives
    verb  <- verbs
    react <- reactions
    return "the #{adj} #{noun} #{verb}ed the code. #{react}"
  end
end
```

```elixir
madlib(
  ["coder", "tester", "hacker", "scorcerer"],
  ["sly", "clever", "crazed"],
  ["fixed", "deleted"],
  ["Hooray!", "Oh no!"]
)
```

```
["the sly coder fixed the code. Hooray!",
 "the sly coder fixed the code. Oh no!",
 "the sly coder deleted the code. Hooray!",
 "the sly coder deleted the code. Oh no!",
 "the clever coder fixed the code. Hooray!",
 "the clever coder fixed the code. Oh no!",
 "the clever coder deleted the code. Hooray!",
 "the clever coder deleted the code. Oh no!",
 "the crazed coder fixed the code. Hooray!",
 "the crazed coder fixed the code. Oh no!",
 "the crazed coder deleted the code. Hooray!",
 "the crazed coder deleted the code. Oh no!",
 "the sly tester fixed the code. Hooray!",
 "the sly tester fixed the code. Oh no!",
 "the sly tester deleted the code. Hooray!",
 "the sly tester deleted the code. Oh no!",
 "the clever tester fixed the code. Hooray!",
 "the clever tester fixed the code. Oh no!",
 "the clever tester deleted the code. Hooray!",
 "the clever tester deleted the code. Oh no!",
 "the crazed tester fixed the code. Hooray!",
 "the crazed tester fixed the code. Oh no!",
 "the crazed tester deleted the code. Hooray!",
 "the crazed tester deleted the code. Oh no!",
 "the sly hacker fixed the code. Hooray!",
 "the sly hacker fixed the code. Oh no!",
 "the sly hacker deleted the code. Hooray!",
 "the sly hacker deleted the code. Oh no!",
 "the clever hacker fixed the code. Hooray!",
 "the clever hacker fixed the code. Oh no!",
 "the clever hacker deleted the code. Hooray!",
 "the clever hacker deleted the code. Oh no!",
 "the crazed hacker fixed the code. Hooray!",
 "the crazed hacker fixed the code. Oh no!",
 "the crazed hacker deleted the code. Hooray!",
 "the crazed hacker deleted the code. Oh no!",
 "the sly scorcerer fixed the code. Hooray!",
 "the sly scorcerer fixed the code. Oh no!",
 "the sly scorcerer deleted the code. Hooray!",
 "the sly scorcerer deleted the code. Oh no!",
 "the clever scorcerer fixed the code. Hooray!",
 "the clever scorcerer fixed the code. Oh no!",
 "the clever scorcerer deleted the code. Hooray!",
 "the clever scorcerer deleted the code. Oh no!",
 "the crazed scorcerer fixed the code. Hooray!",
 "the crazed scorcerer fixed the code. Oh no!",
 "the crazed scorcerer deleted the code. Hooray!",
 "the crazed scorcerer deleted the code. Oh no!"]
```

DO NOTATION IMPLEMENTATION

```elixir
def do_notation(input, chainer) do
  input
  |> normalize()
  |> Enum.reverse()
  |> Witchcraft.Foldable.left_fold(fn
    (continue, {:let, _, [{:=, _, [assign, value]}]}) ->
      quote do: unquote(value) |> fn unquote(assign) -> unquote(continue) end.()

    (continue, {:<-, _, [assign, value]}) ->
      quote do
        import Witchcraft.Chain, only: [>>>: 2]

        unquote(value) >>> (fn unquote(assign) -> unquote(continue) end)
      end

    (continue, value) ->
      quote do
        import Witchcraft.Chain, only: [>>>: 2]
        unquote(value) >>> fn _ -> unquote(continue) end
      end
  end)
end
```

LET'S SEE SOME CODE!

SIMPLE DO-NOTATION USES

WRITER MONAD

Add one to tally
Square number

Need to specify more about type in this case

```elixir
use Witchcraft

exponent =
  fn num ->
    monad writer({0, 0}) do
      tell 1
      return num * num
    end
  end

initial = 42
{result, times} = run(exponent.(initial) >>> exponent >>> exponent)

"#{initial}^#{round(:math.pow(2, times))} = #{result}"

##########
# RESULT #
##########

"42^8 = 9682651996416"
```

Run 3x = num ^ 2 ^ 2 ^ 2

We know how many times this has been run, in a pure fashion

WRITER MONAD

Log initial value

Add a “!” to input and log

Return the result

Run 3 times

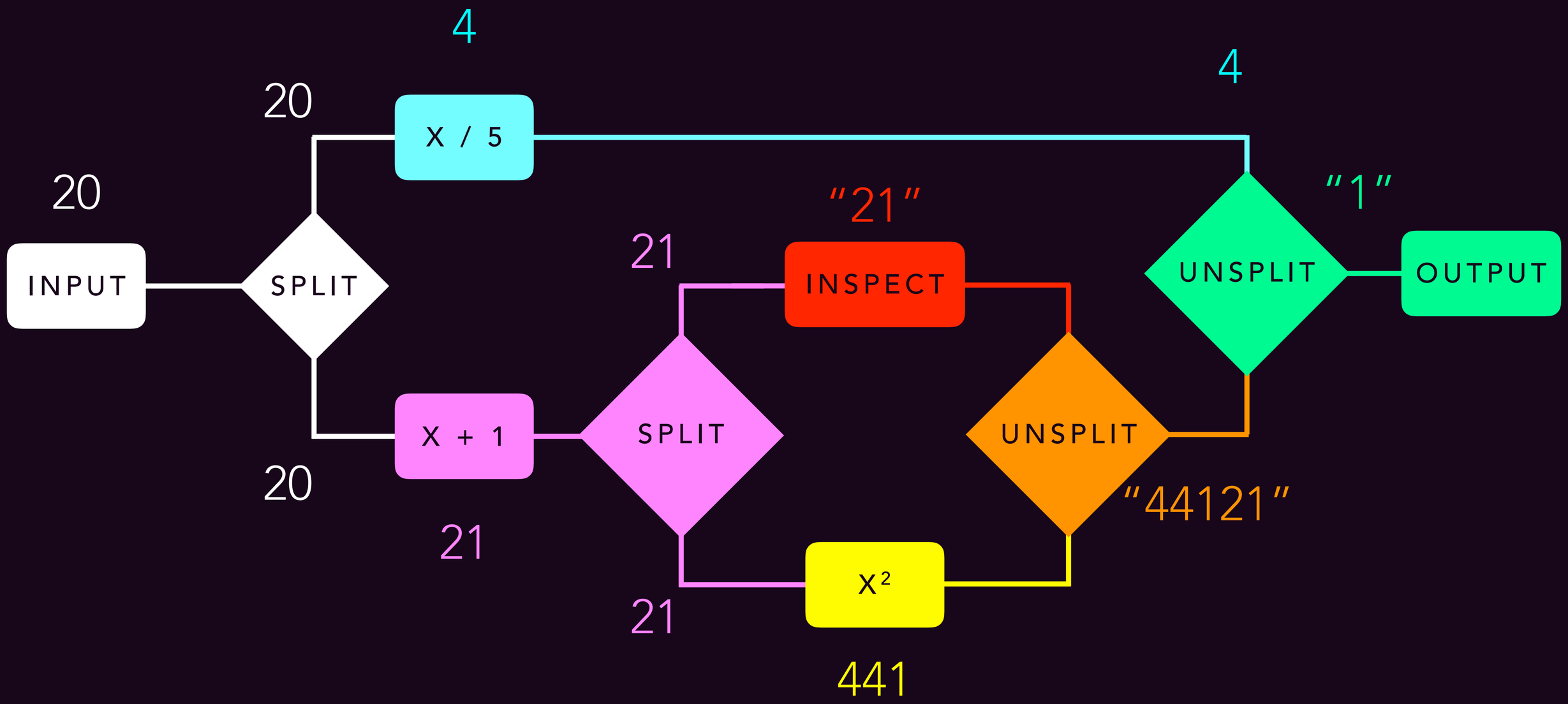
Log of transformations

```elixir
use Witchcraft

excite =
  fn string ->
    monad writer({0.0, "log"}) do
      tell string

      excited <- return "#{string}!"
      tell " => #{excited} ... "

      return excited
    end
  end

{_, logs} =
  "Hi"
  |> excite.()
  >>> excite
  >>> excite
  |> censor(&String.trim_trailing(&1, " ... "))
  |> run()

logs

##########
# RESULT #
##########

"Hi => Hi! ... Hi! => Hi!! ... Hi!! => Hi!!!"
```

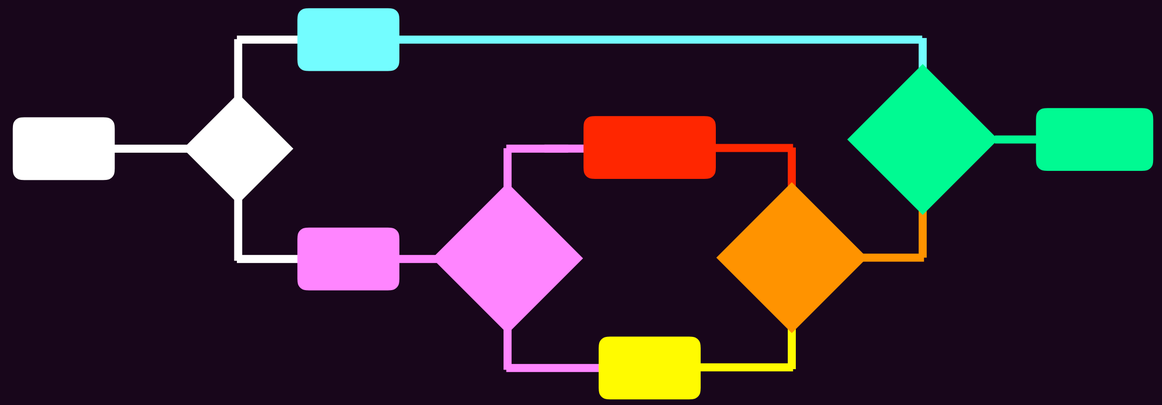
POWERING UP DATA FLOW

ARROWS

ARROWS


4
20
X / 5
4
20
"21"
"1"
21
INPUT
SPLIT
INSPECT
UNSPLIT
OUTPUT
X + 1
SPLIT
UNSPLIT
20
"44121"
21
X²
21
441

ARROWS



```
arrow_diagram =
  fanout(fn x -> x / 5 end, fn y -> y + 1 end
                            <~> (fanout(&inspect/1, fn z -> z * z end))
                            <~> unsplit(fn (a, b) -> "#{b}#{a}" end)
  )
  <~> unsplit(&String.at(&2, round(&1)) end)
```

FUTURE DIRECTIONS

- More ADTs, more type classes
- Pretty printing ADTs
- Automatic deriving
- Alternate minimal definitions (ex. `right_fold` or `fold_map`)
- GenArrow

THANK YOU

- hex.pm/packages/witchcraft
- hex.pm/packages/quark
- hex.pm/packages/algae
- hex.pm/packages/type_class
- @expede
- brooklyn@robotoverlord.io