LAMBDACONF 2016

WITCHCRAFT

WHERE ARE `FLIP`, `ID`, `CONST`, `FIX`, AND ALL OF THE OTHER FUNCTIONAL NICETIES THAT WE'RE USED TO?

A variation on the Bulb Paradox

HOW CAN I MAKE THIS MORE LIKE HASKELL?

A variation on the Bulb Paradox

WITCHCRAFT [WICH-KRAFT, -KRAHFT] NOUN

1. The practice of, and belief in, magical skills and abilities that are able to be exercised by persons with the necessary esoteric secret knowledge

2. A math-/algebra-/category-inspired library for Elixir

A LOT OF PEOPLE APPROACH MATH-Y ABSTRACTIONS AS IF THEY WERE DARK MAGIC

THIS DOESN'T HAVE TO BE THE CASE!


ABSTRACTIONS,
ABSTRACTIONS EVERYWHERE

THE PURPOSE OF ABSTRACTION IS NOT TO BE VAGUE, BUT TO CREATE A NEW SEMANTIC LEVEL IN WHICH ONE CAN BE ABSOLUTELY PRECISE

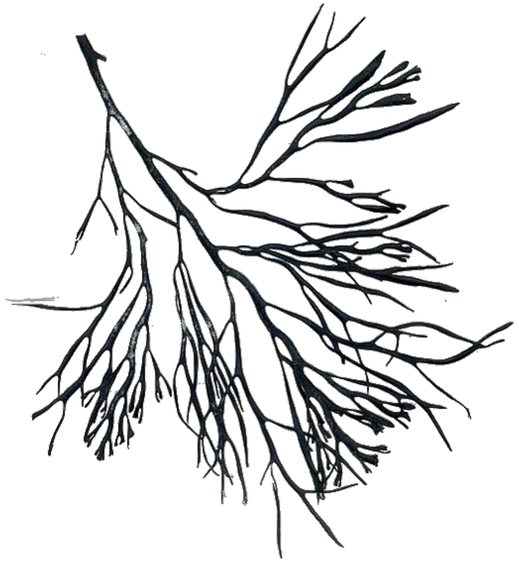
Edsger W. Dijkstra

PRECISION

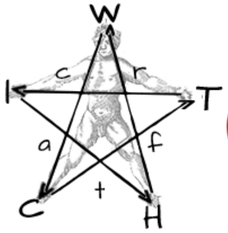
- Well-understood algebraic properties make us *free of edge cases*
- Can be thought of as “functional design patterns”
- Leverage the past 60+ years of progress
- Port patterns from other languages

Algae

Bootstrapped
algebraic data types
for Elixir



Witchcraft



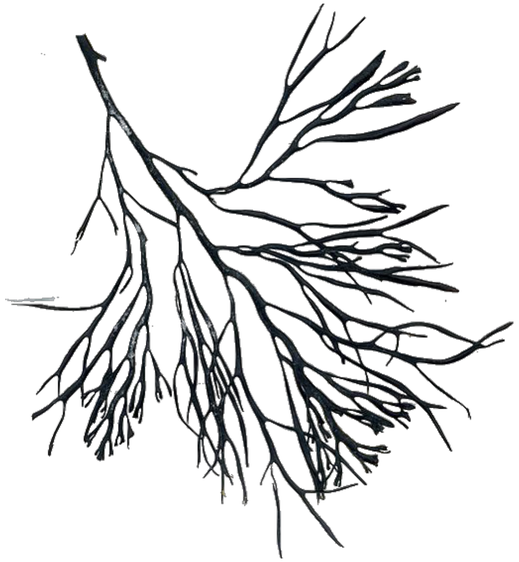
A category library for Elixir

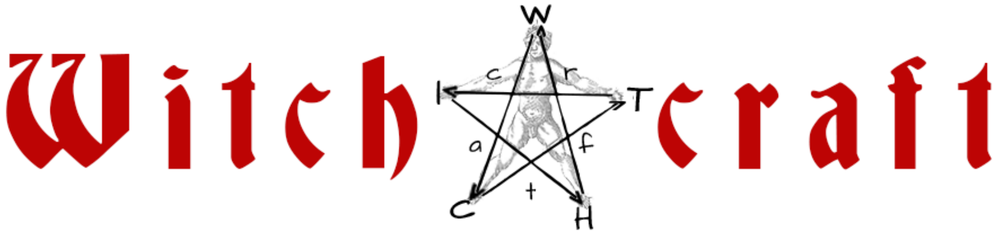
THREE LIBRARIES (BASICALLY "ELIXIRZ")

- Quark
  - github.com/robot-overlord/quark
  - Common combinators & functional helpers
    - SKI, currying, &c
- Algae
  - github.com/robot-overlord/algae
  - Common datatypes
    - Maybe, Either, Tree, Free, &c
- Witchcraft
  - github.com/robot-overlord/witchcraft
  - Common algebras
    - Functor, applicative, monad, arrow, &c

Algae

Bootstrapped
algebraic data types
for Elixir





VALUES

- Lower barrier to entry
- Consistency
- Pragmatism
- Compatibility / idiomatic
- Pedagogy

WHAT DOES IT GET US? (HIGH LEVEL)

- Vertical and horizontal, or strategy and tactics
- Generalization makes for highly reusable code
- Write really (computationally) semantic code
- Less stuff to test
  - The abstraction portion can be tested separately, but used in many scenarios
- Fun!
  - Porting stuff and using patterns from other languages

HANDY COMBINATORS, LIKE FIXED-POINTS

```
factorial =
  fn (0,_) -> 1
     (1,_) -> 1
     (n, fun) -> n * fun.(n - 1, fun)
end
```

Manual recursion, common pattern, can be abstracted out

```
fac = fn fac ->
  fn 0 -> 0
     1 -> 1
     n -> n * fac.(n - 1)
  end
end

factorial = fix fac

iex> factorial.(9)
362880
```

Only the “guts” of the calculation

Pass to `fix` to get the same result

BONUS: nice to test each the decoupled inner logic, as is simpler (no recursion)

HANDY ABSTRACTIONS, LIKE APPLICATIVE FUNCTORS

- Works on any datatype in the protocol
- Guarantees (more on this later)
  - Always works as expected
  - Fewer things to test / tests included
- Operator and function notations
  - Choose of data flow direction
- Semantic behaviour for each datatype
  - List can represent “nondeterminism”
  - Maybe can represent presence/emptiness

```
# Applicative list
## Example helpers
add_one = &(&1 + 1)
times_ten = &(&1 * 10)
prod = fn x -> (fn y -> x * y end) end

[1,2,3] ~>> [add_one, times_ten]
#=> [2,3,4,10,20,30]

[9, 10] ~>> ([1,2,3] ~> prod)
#=> [9, 10, 18, 20, 27, 30]

prod <~ [1,2,3] <<~ [9, 10]
#=> [9, 10, 18, 20, 27, 30]
```

LEVERAGE PROPERTIES

- ex. Monoids guarantee identity and associativity
- Could do something like break up a complex problem over many processes
  - As long as the parts are reassembled in the correct order eventually, the actual joining can happen in any order
  - Could ignore identity inputs
  - Don’t have to worry about “wrong type” errors
- Automagically works on any datatype defined for `Witchcraft.Monoid`
- This hypothetical application could now be a reusable library for any monoid!

```
# Binary combining operation <|>,
# always returns same kind of monoid
@spec MType <|> MType :: MType

# Identity
identity <|> x = x
x <|> identity = x

# Associative
 a <|>  b  <|> c
(a <|>  b) <|> c
 a <|> (b  <|> c)
```

WHAT ARE THE DOWNSIDES?

- Higher barrier to entry
  - Puts more burden on the programmer up-front, delayed gratification
    - ie: need to learn concepts, patterns, ways of thinking, &c
  - Similar to how in OO, everyone learns the Gang of Four patterns
  - Some abstractions can be mind-bending at first
- Attempting to mitigate this as much as possible with explanatory docs, examples, and doctests

DIFFERING WAYS
|> OF THINKING
<<~ ARE COMPATIBLE

BREAKING THE MODEL

- Some of this requires currying by default for generality
  - Lots of partial application
  - But Elixir is an arity-based language!
- Elixir code is often pretty concrete
- Bootstrapping into Elixir relies on additional tools
  - ex. Dialyzer, QuickCheck

DIRECTIONALITY

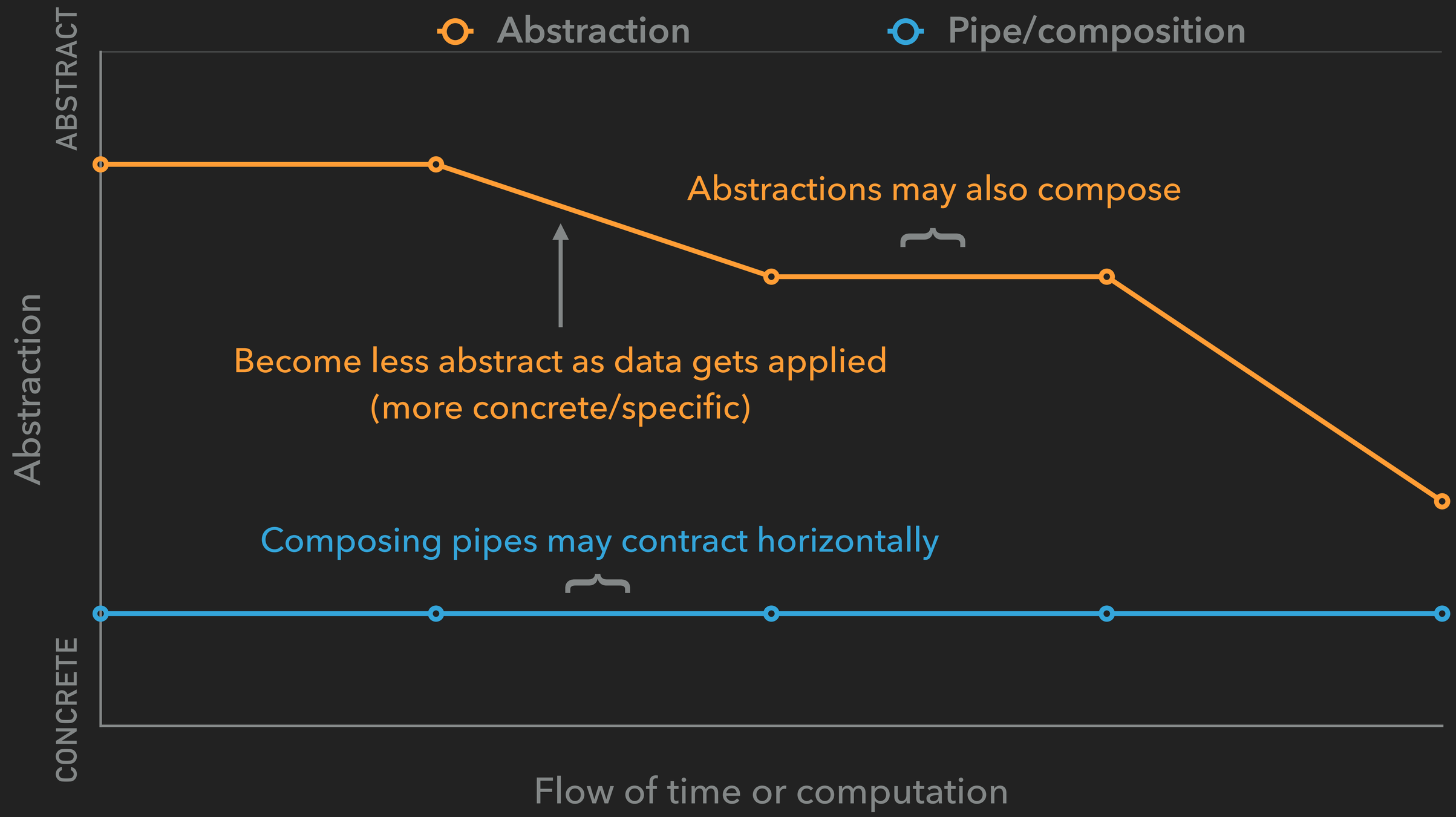
- One characteristic of FP structural solutions or being data-oriented
- In Elixir, we generally think from linearly, left-to-right
  - “forward”
  - “flow”
- In maths, LISPs, and MLs, we often think multi-dimensionally
  - Property-based (ex. monoid)
  - Point-free style

LINEARITY

- Data flow is easy to think of as being linear
- Abstract functions are (often) also composable
- Can think of abstractions as being the vertical to data flow's horizontal

STRATEGY VS TACTICS

- Big-picture vs details
- Decouple thinking in the large from specifics (choosing protocol vs defimpl)
- Broad class of problem vs specific problem


Abstraction
Pipe/composition
ABSTRACT
Abstraction
CONCRETE
Abstractions may also compose
Become less abstract as data gets applied
(more concrete/specific)
Composing pipes may contract horizontally
Flow of time or computation

BECOMES LESS ABSTRACT?

- Simple example:
  - `fold` is more general than `map`
  - **Elixir:** `Enum.map(list, f) == right_reduce(list, &([f.(&1) | &2]))`
  - **Haskell:** `fmap == foldr (λx xs → f x : xs) []`
- `More complex example (applicative function akin to scanning)`
  - **Witchcraft:** `&(&1 + &2) <~ foo ≪~ bar`
  - **Haskell:** `(+) <$> foo <*> bar`

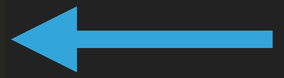
LET'S TALK ABOUT

TYPES

PORTING ADTS

SEVERAL ATTEMPTS TO PORT COMMON DATATYPES TO ELIXIR (“MAYBE” IS SHOWN)

```elixir
{:just, "something"}
# OR
{:error}
```



Idiomatic Elixir, but not as general as could be

```elixir
defmodule Maybe do
  defstruct just: nil, nothing: false
end


# But what about this case?
%Maybe{just: "something", nothing: true}
```

Nothing ≠ :error

SURE, ELIXIR DOESN'T HAVE ADTS

BUT WE CAN FAKE IT WITH STRUCTS

```elixir
defmodule Algae.Maybe do
  use Quark.Partial

  @type t :: Just.t | Nothing.t

  defmodule Nothing do
    @type t :: %Nothing{}
    defstruct []
  end

  defmodule Just do
    @type t :: %Just{just: any}
    defstruct [:just]
  end

  # Convenience functions
end
```

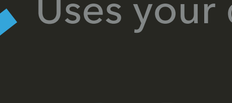
ELIXIR CAN'T ENFORCE PROPERTIES

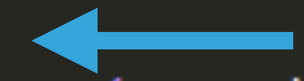
BUT IT HAS PROTOCOLS

```elixir
defprotocol Witchcraft.Applicative do
  # Docs

  @fallback_to_any true

  @spec wrap(any, any) :: any
  def wrap(specimen, bare)

  @spec seq(any, (... -> any)) :: any
  def seq(wrapped_value, wrapped_function)
end

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

  def wrap(_, bare), do: %Algae.Id{id: bare}
  def seq(%Id{id: value}, %Id{id: fun}), do: %Id{id: curry(fun).(value)}
end
```

The only custom code for this data type
(low effort required)

```elixir
defmodule Witchcraft.Applicative.Property do
  # Docs & imports

  @spec spotcheck_identity(any) :: boolean
  def spotcheck_identity(value), do: (value ~>> wrap(value, &id/1)) == value

  @spec spotcheck_composition(any, any, any) :: boolean
  def spotcheck_composition(value, fun1, fun2) do
    wrap(value, &compose/2) <<~ fun1 <<~ fun2 <<~ value == fun1 <<~ (fun2 <<~ value)
  end

  @spec spotcheck_homomorphism(any, any, fun) :: boolean
  def spotcheck_homomorphism(specemin, val, fun) do
    curried = curry(fun)
    wrap(specemin, val) ~>> wrap(specemin, curried) == wrap(specemin, curried.(val))
  end

  def spotcheck_interchange(bare_val, wrapped_fun) do
    wrap(wrapped_fun, bare_val) ~>> wrapped_fun
      == wrapped_fun ~>> wrap(wrapped_fun, &(bare_val |> curry(&1).()))
  end

  @spec spotcheck_functor(any, fun) :: boolean
  def spotcheck_functor(wrapped_value, fun) do
    wrapped_value ~> fun == wrapped_value ~>> wrap(wrapped_value, fun)
  end
end
```



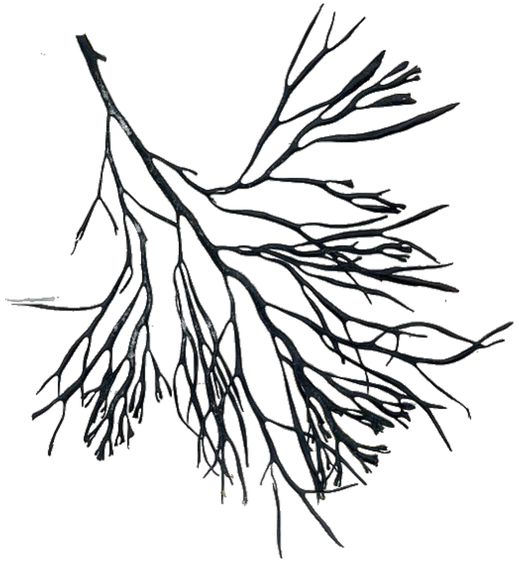


SUMMARY

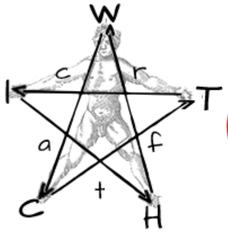
- Abstraction is a handy tool in Elixir
- Pipes are compatible with other abstractions
- Thinking
  - 2-dimensionally
  - With properties
- Often have to break the arity model
- Can “fake” a lot of ADTs in Elixir

Algae

Bootstrapped
algebraic data types
for Elixir



Witchcraft



A category library for Elixir

THE END

- Please contribute, open feature requests, and so on!
  - github.com/robot-overlord/quark
  - github.com/robot-overlord/algae
  - github.com/robot-overlord/witchcraft
- Get in touch :)
  - bez@brooklynzelenka.com
  - @expede
  - medium.com/@expede
  - github.com/expede