THE UNREASONABLE EFFECTIVENESS OF *MONADS*

🔥🚀 AUTOMATION, STRUCTURE, & PURELY FUNCTIONAL EFFECTS ⚙️✨

Simplicity is prerequisite for **reliability**

EDSGER DIJKSTRA

THE UNREASONABLE EFFECTIVENESS OF MONADS

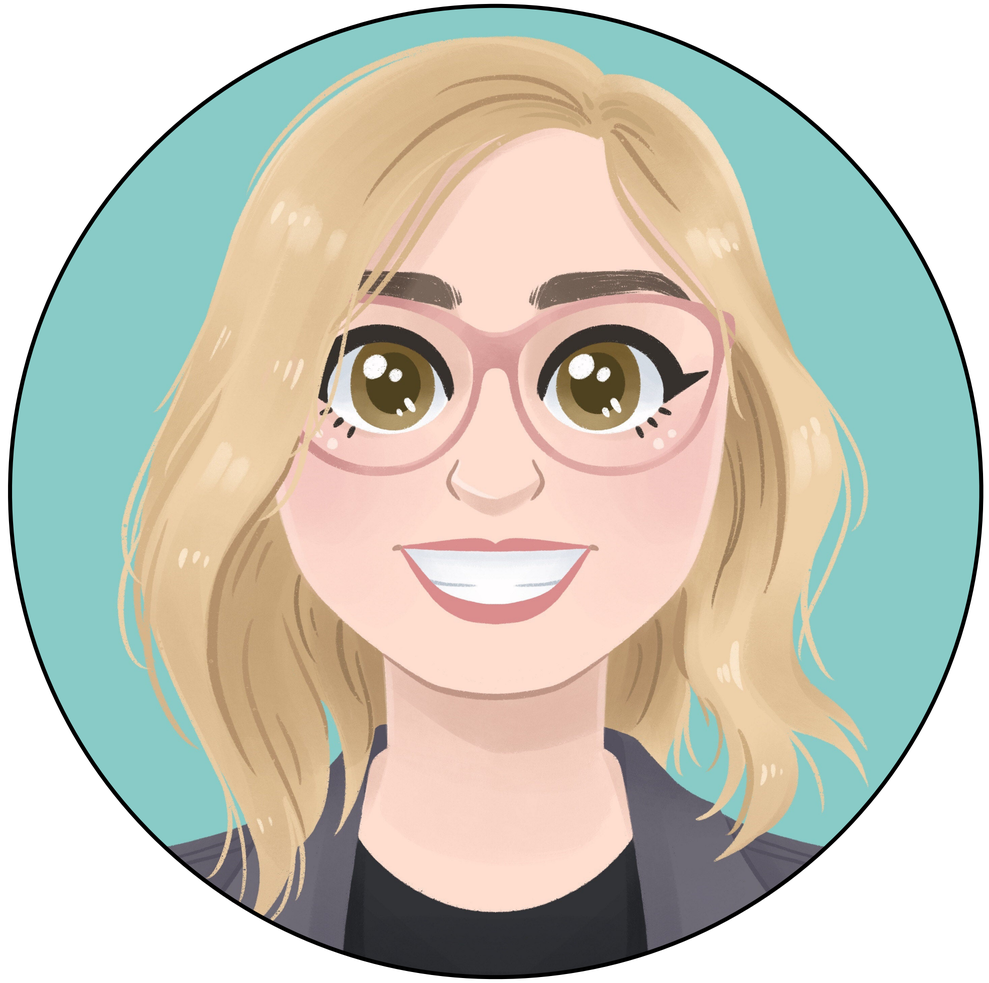
BROOKLYN ZELENKA, @expede

fission

THE UNREASONABLE EFFECTIVENESS OF MONADS

BROOKLYN ZELENKA, @expede

- Cofounder/CTO at Fission
  - `https://fission.codes`
- PLT & VM enthusiast
- Previously an Ethereum Core Dev
- Primary author of Witchcraft Suite
- Used to teach Elixir professionally
- Now with a Haskell team 😉



fission

THE UNREASONABLE EFFECTIVENESS OF MONADS

SALES PITCH

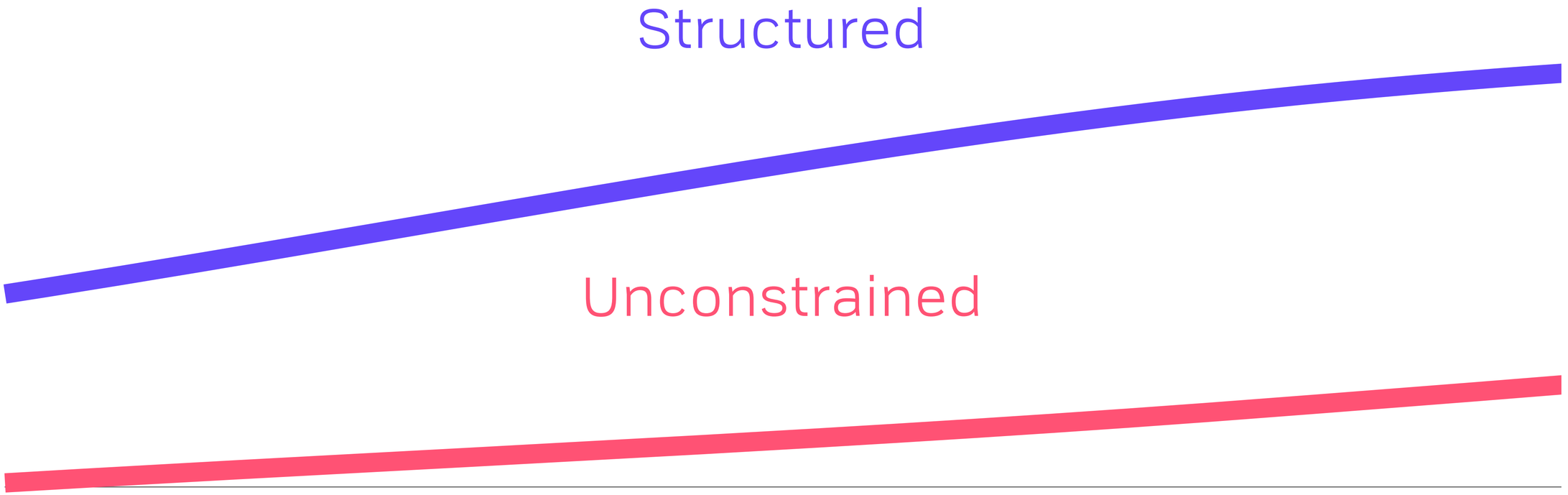
THE UNREASONABLE EFFECTIVENESS OF MONADS

SALES PITCH

1. Broad trend towards functional techniques
2. Handle increasing complexity
3. Familiar != “simple”
4. You’re already sitting in the “polyglot and fringe” track 😉

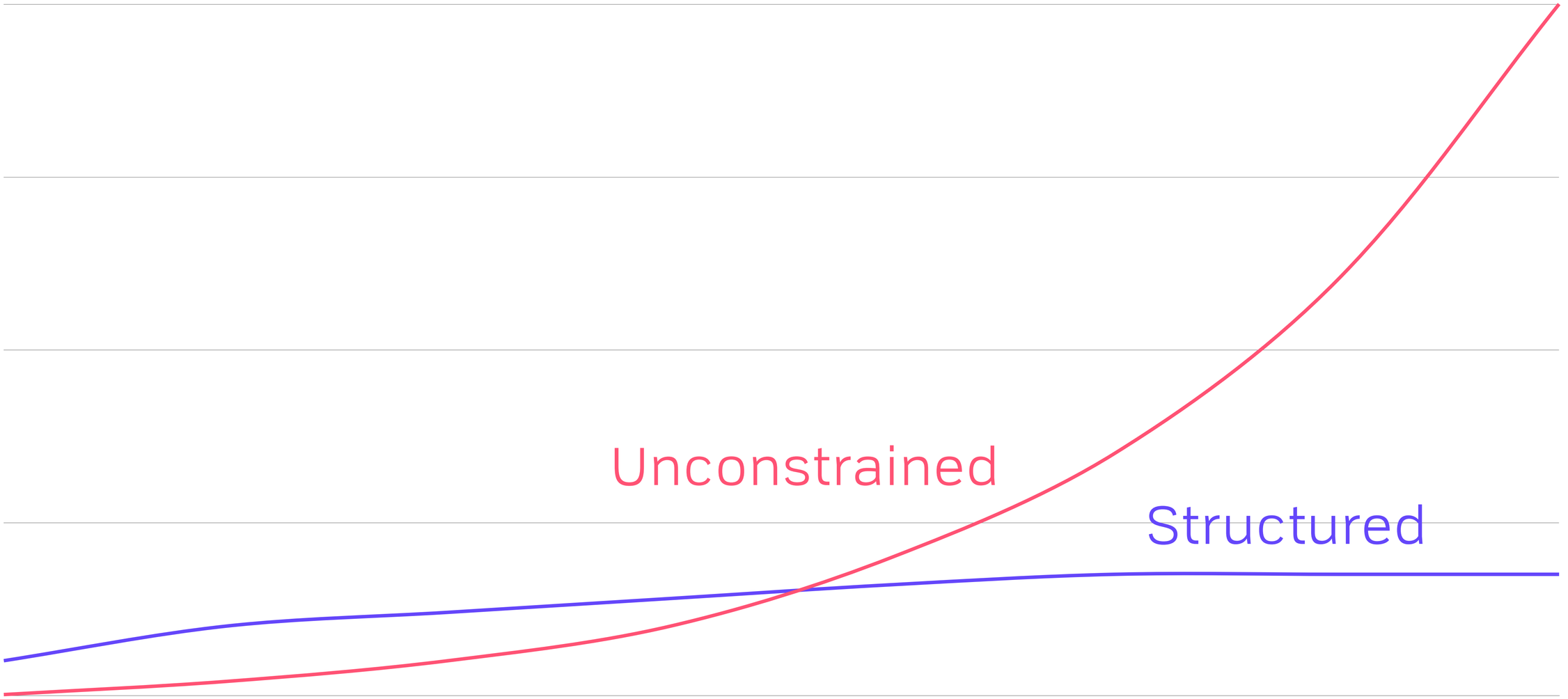
THE UNREASONABLE EFFECTIVENESS OF MONADS

EDUCATION INVESTMENT VS COMPLEXITY



THE UNREASONABLE EFFECTIVENESS OF MONADS

EDUCATION INVESTMENT VS COMPLEXITY



THE UNREASONABLE EFFECTIVENESS OF MONADS

EFFECT-FOCUSED

THE UNREASONABLE EFFECTIVENESS OF MONADS

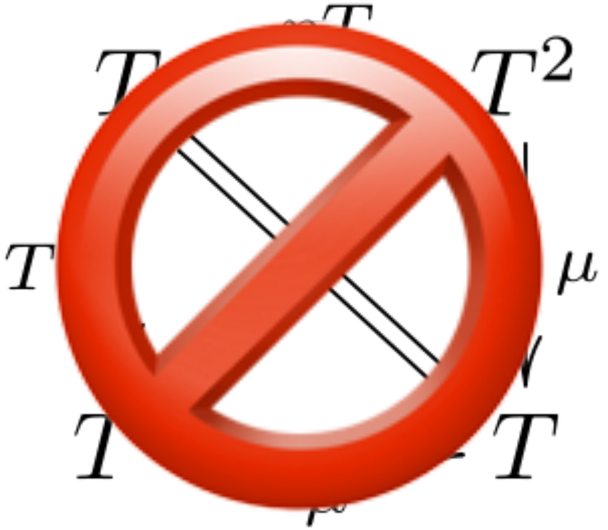
EFFECT-FOCUSED

- Monads are extremely well defined

THE UNREASONABLE EFFECTIVENESS OF MONADS

EFFECT-FOCUSED

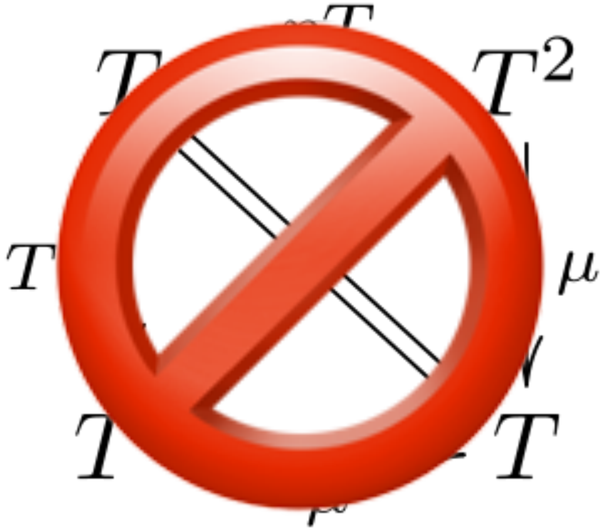
- Monads are extremely well defined
- Steer away from the math



THE UNREASONABLE EFFECTIVENESS OF MONADS

EFFECT-FOCUSED

- Monads are extremely well defined
- Steer away from the math
- Many uses, but the main practical one is **effects**



THE UNREASONABLE EFFECTIVENESS OF MONADS

TABLE OF CONTENTS

THE UNREASONABLE EFFECTIVENESS OF MONADS

TABLE OF CONTENTS

STRUCTURED ABSTRACTION

STRUCTURED ABSTRACTION

STRUCTURED ABSTRACTION

structure
/ 'strʌktʃər /

1. A mode of **building**, construction, or **organization**; arrangement of parts, elements, or constituents

   e.g. a pyramidal structure.

STRUCTURED ABSTRACTION

abstraction
/ æb'stræk∫ən /

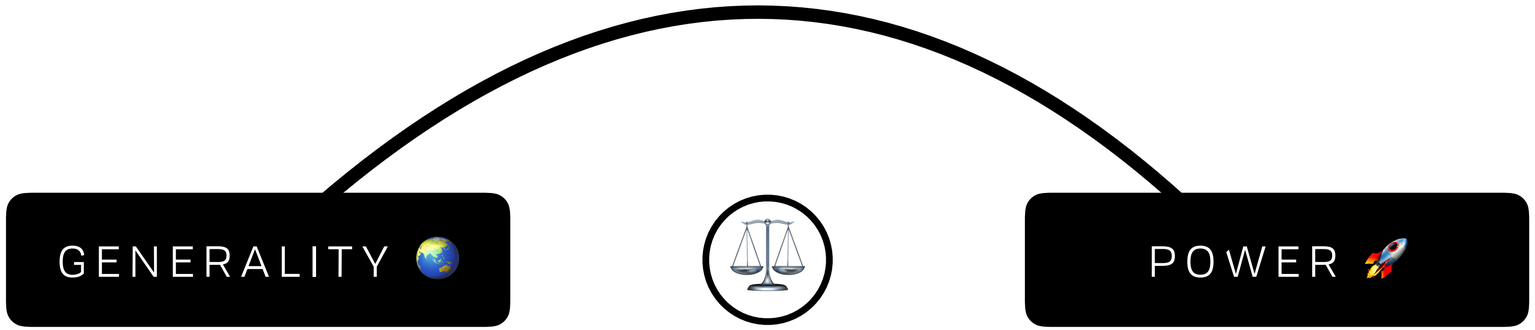
1. Something that concentrates in itself the **essential qualities** of anything **more extensive** or more general, or **of several things**; its essence.

STRUCTURED ABSTRACTION
TRADE-OFFS

- “GOTOs considered harmful”
- Exchange control for understanding

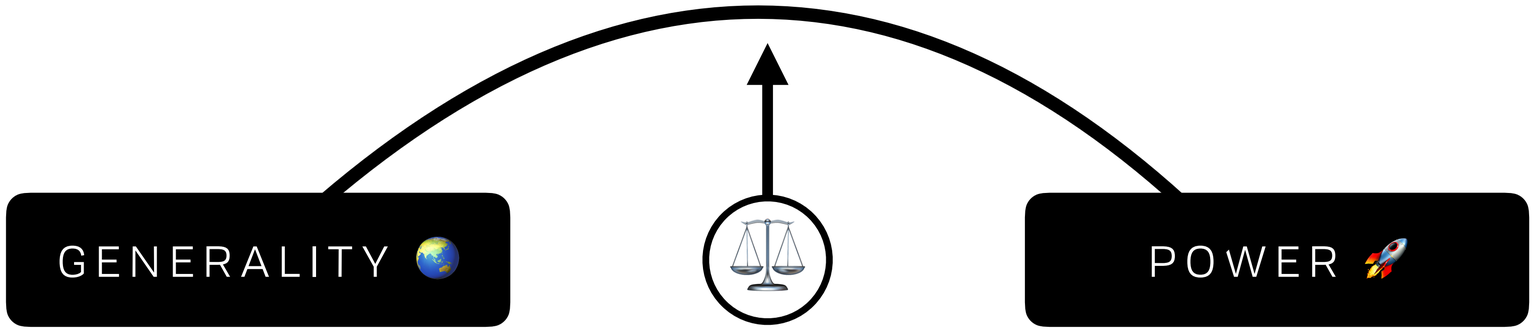
STRUCTURED ABSTRACTION
TRADE-OFFS

- “GOTOs considered harmful”
- Exchange control for understanding



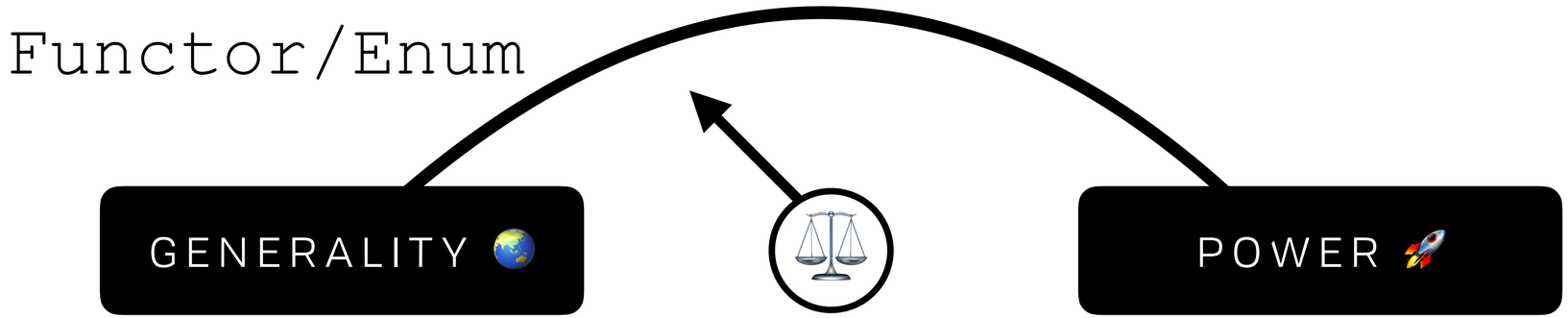
STRUCTURED ABSTRACTION
TRADE-OFFS

- “GOTOs considered harmful”
- Exchange control for understanding



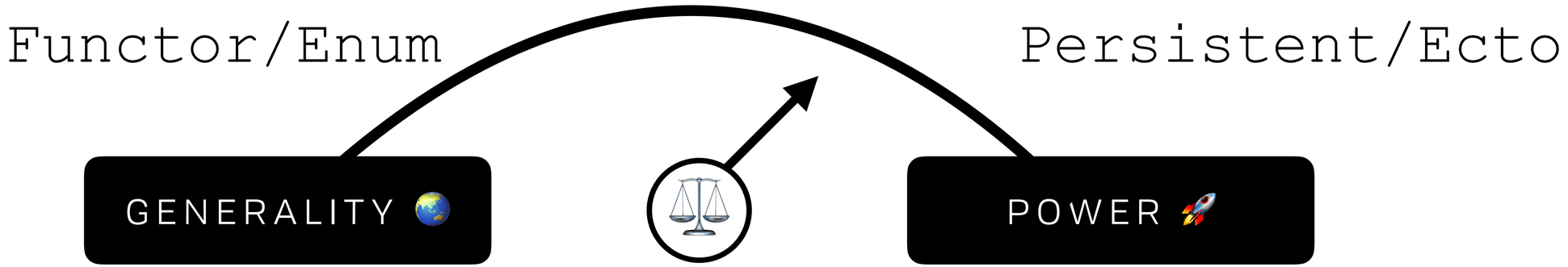
STRUCTURED ABSTRACTION
TRADE-OFFS

- “GOTOs considered harmful”
- Exchange control for understanding



STRUCTURED ABSTRACTION
TRADE-OFFS

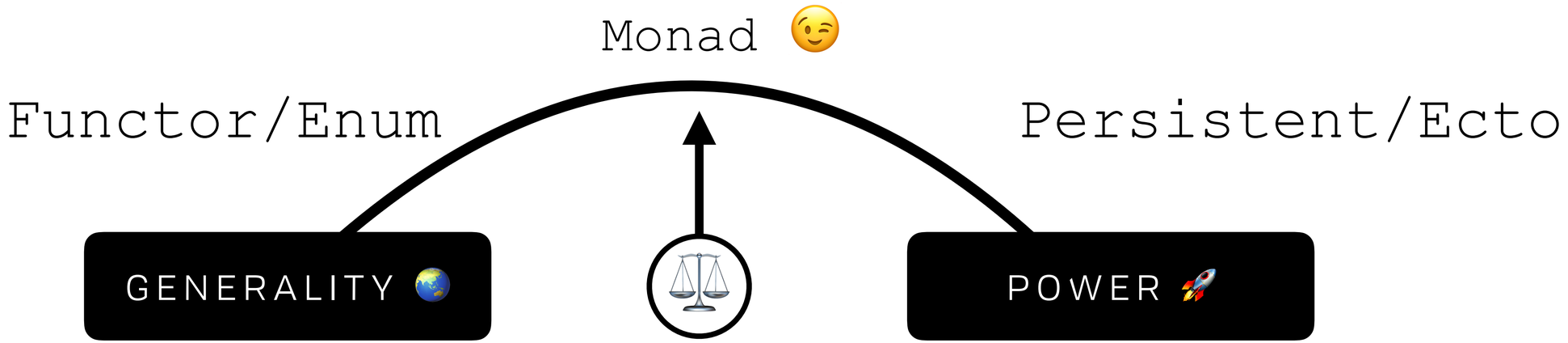
- “GOTOs considered harmful”
- Exchange control for understanding



STRUCTURED ABSTRACTION

TRADE-OFFS

- “GOTOs considered harmful”
- Exchange control for understanding



STRUCTURED ABSTRACTION

SIMPLE EXAMPLE: **SEMIGROUP**

STRUCTURED ABSTRACTION

SIMPLE EXAMPLE: **SEMIGROUP**

- Not a data structure

STRUCTURED ABSTRACTION

SIMPLE EXAMPLE: **SEMIGROUP**

- Not a data structure
- Not a function

STRUCTURED ABSTRACTION

SIMPLE EXAMPLE: **SEMIGROUP**

- Not a data structure
- Not a function
- An interface & rules!

STRUCTURED ABSTRACTION

SIMPLE EXAMPLE: **SEMIGROUP**

- Not a data structure
- Not a function
- An interface & rules!

```
(a • b) • c == a • (b • c)

AKA

concat(concat(a, b), c) == concat(a, concat(b, c))
```

STRUCTURED ABSTRACTION

A SEMIGROUP ON…

STRUCTURED ABSTRACTION

A SEMIGROUP ON…

```haskell
class Semigroup a where
  concat :: a -> a -> a
```

```elixir
defprotocol Semigroup do
  def concat(a, b)
end
```

STRUCTURED ABSTRACTION

A SEMIGROUP ON…

```haskell
class Semigroup a where
  concat :: a -> a -> a

instance Semigroup Int where
  concat a b = a + b
```

```elixir
defprotocol Semigroup do
  def concat(a, b)
end

defimpl Semigroup, for: Integer do
  def concat(a, b), do: a + b
end
```

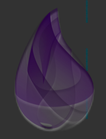
```javascript
Number.prototype.concat = function (num) {
  return this.valueOf() + num;
};
```

JS

STRUCTURED ABSTRACTION

A SEMIGROUP ON…

```haskell
class Semigroup a where
  concat :: a -> a -> a

instance Semigroup Int where
  concat a b = a + b

instance Semigroup [a] where
  concat xs ys = xs ++ ys
```



```elixir
defprotocol Semigroup do
  def concat(a, b)
end

defimpl Semigroup, for: Integer do
  def concat(a, b), do: a + b
end

defimpl Semigroup, for: List do
  def concat(xs, ys), do: xs ++ ys
end
```

```javascript
Number.prototype.concat = function (num) {
  return this.valueOf() + num;
};

// Array already has a concat function
// that does what we want 👍
```

STRUCTURED ABSTRACTION

UNLAWFUL COUNTEREXAMPLE 🚨

```
 1.0 / (2.0  / 3.0) == 1.5
(1.0 /  2.0) / 3.0  == 0.1666...
```

STRUCTURED ABSTRACTION

WE USE LOTS OF DIFFERENT FEATURES *DAILY*

STRUCTURED ABSTRACTION

WE USE LOTS OF DIFFERENT FEATURES *DAILY*

- Promises (AKA async/await)
  - Network
  - Database
  - Long computation

```
const result     = await runProcess();
const nextResult = await doThing(result);
const moreResult = await nextFunc(nextResult);
const final      = await moreFunc(moreResult);
```

STRUCTURED ABSTRACTION

WE USE LOTS OF DIFFERENT FEATURES *DAILY*

- Promises (AKA async/await)
  - Network
  - Database
  - Long computation
- throw/catch

```
const result     = await runProcess();
const nextResult = await doThing(result);
const moreResult = await nextFunc(nextResult);
const final      = await moreFunc(moreResult);
```

```
try {
  explodingFunc();
} catch(error) {
  handleOrReport(error);
}
```

STRUCTURED ABSTRACTION

WE USE LOTS OF DIFFERENT FEATURES *DAILY*

- Promises (AKA async/await)
  - Network
  - Database
  - Long computation
- throw/catch
- Tracing

```
const result     = await runProcess();
const nextResult = await doThing(result);
const moreResult = await nextFunc(nextResult);
const final      = await moreFunc(moreResult);
```

```
function foo() {
  console.trace();

  function bar() {
    console.trace();
  }

  bar();
}

foo();
```

```
try {
  explodingFunc();
} catch(error) {
  handleOrReport(error);
}
```

STRUCTURED ABSTRACTION

WE USE LOTS OF DIFFERENT FEATURES *DAILY*

- Promises (AKA async/await)
  - Network
  - Database
  - Long computation
- throw/catch
- Tracing
- Context or config values

```
const result     = await runProcess();
const nextResult = await doThing(result);
const moreResult = await nextFunc(nextResult);
const final      = await moreFunc(moreResult);
```

```
function foo() {
  console.trace();

  function bar() {
    console.trace();
  }

  bar();
}

foo();
```

```
try {
  explodingFunc();
} catch(error) {
  handleOrReport(error);
}
```

```
export const themes = {
  // theme data
};

export const ThemeContext = React.createContext(
  themes.dark // default value
);

// ...

let theme = this.context;
```

STRUCTURED ABSTRACTION

BUT MAYBE NOT SO DIFFERENT

```
try {
  explodingFunc();
} catch(error) {
  handleOrReport(error);
}
```

```
const result     = await runProcess();
const nextResult = await doThing(result);
const moreResult = await nextFunc(nextResult);
const final      = await moreFunc(moreResult);
```

```
function foo() {
  console.trace();

  function bar() {
    console.trace();
  }

  bar();
}

foo();
```

```
runProcess
  .then(result => {
    return doThing(result).then(nextResult => {
      return nextFunc(nextResult).then(moreResult => {
        return moreFunc(moreResult);
      });
    });
  });
```

STRUCTURED ABSTRACTION

BUT MAYBE NOT SO DIFFERENT

```
try {
  explodingFunc();
} catch(error) {
  handleOrReport(error);
}
```

```
const result     = await runProcess();
const nextResult = await doThing(result);
const moreResult = await nextFunc(nextResult);
const final      = await moreFunc(moreResult);
```

```
runProcess
  .then(result => {
    return doThing(result).then(nextResult => {
      return nextFunc(nextResult).then(moreResult => {
        return moreFunc(moreResult);
      });
    });
  });
```

```
function foo() {
  console.trace();

  function bar() {
    console.trace();
  }

  bar();
}

foo();
```

STRUCTURED ABSTRACTION

BUT MAYBE NOT SO DIFFERENT

```
try {
  explodingFunc();
} catch(error) {
  handleOrReport(error);
}
```

```
const result     = await runProcess();
const nextResult = await doThing(result);
const moreResult = await nextFunc(nextResult);
const final      = await moreFunc(moreResult);
```

```
runProcess
  .then(result => {
    return doThing(result).then(nextResult => {
      return nextFunc(nextResult).then(moreResult => {
        return moreFunc(moreResult);
      });
    });
  });
```

```
function foo() {
  console.trace();

  function bar() {
    console.trace();
  }

  bar();
}

foo();
```

STRUCTURED ABSTRACTION

BUT MAYBE NOT SO DIFFERENT

```
try {
  explodingFunc();
} catch(error) {
  handleOrReport(error);
}
```

```
const result     = await runProcess();
const nextResult = await doThing(result);
const moreResult = await nextFunc(nextResult);
const final      = await moreFunc(moreResult);
```

```
runProcess
  .then(result => {
    return doThing(result).then(nextResult => {
      return nextFunc(nextResult).then(moreResult => {
        return moreFunc(moreResult);
      });
    });
  });
```

```
function foo() {
  console.trace();

  function bar() {
    console.trace();
  }

  bar();
}

foo();
```

STRUCTURED ABSTRACTION

BUT MAYBE NOT SO DIFFERENT

```
try {
  explodingFunc();
} catch(error) {
  handleOrReport(error);
}
```

```
const result      = await runProcess();
const nextResult  = await doThing(result);
const moreResult  = await nextFunc(nextResult);
const final       = await moreFunc(moreResult);
```

```
runProcess
  .then(result => {
    return doThing(result).then(nextResult => {
      return nextFunc(nextResult).then(moreResult => {
        return moreFunc(moreResult);
      });
    });
  });
```

```
function foo() {
  console.trace();

  function bar() {
    console.trace();
  }

  bar();
}

foo();
```

STRUCTURED ABSTRACTION

BUT MAYBE NOT SO DIFFERENT

```
try {
  explodingFunc();
} catch(error) {
  handleOrReport(error);
}
```

```
const result    = await runProcess();
const nextResult = await doThing(result);
const moreResult = await nextFunc(nextResult);
const final     = await moreFunc(moreResult);
```

```
runProcess
  .then(result => {
    return doThing(result).then(nextResult => {
      return nextFunc(nextResult).then(moreResult => {
        return moreFunc(moreResult);
      });
    });
  });
```

```
function foo() {
  console.trace();

  function bar() {
    console.trace();
  }

  bar();
}

foo();
```

STRUCTURED ABSTRACTION

BUT MAYBE NOT SO DIFFERENT

```
const result    = await runProcess();
const nextResult = await doThing(result);
const moreResult = await nextFunc(nextResult);
const final     = await moreFunc(moreResult);
```

```
runProcess
  .then(result => {
    return doThing(result).then(nextResult => {
      return nextFunc(nextResult).then(moreResult => {
        return moreFunc(moreResult);
      });
    });
  });
```

```
try {
  explodingFunc();
} catch(error) {
  handleOrReport(error);
}
```

```
function foo() {
  console.trace();

  function bar() {
    console.trace();
  }

  bar();
}

foo();
```

STRUCTURED ABSTRACTION

BUT MAYBE NOT SO DIFFERENT

```
try {
  explodingFunc();
} catch(error) {
  handleOrReport(error);
}
```

```
const result    = await runProcess();
const nextResult = await doThing(result);
const moreResult = await nextFunc(nextResult);
const final     = await moreFunc(moreResult);
```

```
runProcess
  .then(result => {
    return doThing(result).then(nextResult => {
      return nextFunc(nextResult).then(moreResult => {
        return moreFunc(moreResult);
      });
    });
  });
```

```
function foo() {
  console.trace();

  function bar() {
    console.trace();
  }

  bar();
}

foo();
```

YOU'RE ALREADY DOING THIS

YOU'RE ALREADY DOING THIS

🤾 SURPRISE ATTACK EFFECTS 💥

FRONT EFFECTS

FRONT EFFECTS

FRONT EFFECTS
SIDE EFFECTS

FRONT EFFECTS

SIDE EFFECTS

- Implicit effects that happen “off to the side”

FRONT EFFECTS

SIDE EFFECTS

- Implicit effects that happen “off to the side”
- Built into the language / platform

FRONT EFFECTS

SIDE EFFECTS

- Implicit effects that happen “off to the side”
- Built into the language / platform
- Hard to inspect

FRONT EFFECTS

SIDE EFFECTS

- Implicit effects that happen “off to the side”
- Built into the language / platform
- Hard to inspect
  - …thus hard to change, compose, or test

FRONT EFFECTS

EFFECTS-AS-DATA

FRONT EFFECTS

EFFECTS-AS-DATA

- Data is simple

FRONT EFFECTS

EFFECTS-AS-DATA

- Data is simple
- Effects don’t need to be separate: **express effects as data**

FRONT EFFECTS

EFFECTS-AS-DATA

- Data is simple
- Effects don’t need to be separate: **express effects as data**
- Write your own effects

FRONT EFFECTS

EFFECTS-AS-DATA

- Data is simple
- Effects don’t need to be separate: **express effects as data**
- Write your own effects
- Inspect them as needed

FRONT EFFECTS

EFFECTS-AS-DATA

- Data is simple
- Effects don’t need to be separate: **express effects as data**
- Write your own effects
- Inspect them as needed
- Compose as needed

FRONT EFFECTS

ENTER THE MONAD

FRONT EFFECTS

ENTER THE MONAD

- A common interface to make this straightforward!

FRONT EFFECTS

ENTER THE MONAD

- A common interface to make this straightforward!
- Learn once, use everywhere

THE FUNCTOR TOWER

THE FUNCTOR TOWER

LET'S COVER SOME INTERNALS

Progress is possible only if we train ourselves to think about programs **without** thinking of them as pieces of **executable code**

EDSGER DIJKSTRA

THE FUNCTOR TOWER

FUNCTOR

THE FUNCTOR TOWER

FUNCTOR

- Always returns the same shape!

THE FUNCTOR TOWER

FUNCTOR

`Functor.map`

- Always returns the same shape!

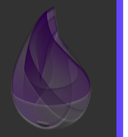
THE FUNCTOR TOWER
FUNCTOR

Functor.map

- Always returns the same shape!

```haskell
class Functor f where
  fmap :: (a -> b) -> f a -> f b

instance Functor [] where
  fmap f []       = []
  fmap f (x : xs) = f x : fmap f xs
```

```elixir
defimpl Functor, for: List do
  def map([],         func), do: []
  def map([x | xs], func) do
    [func.(x) | map(xs, func)]
end
```



```javascript
[1,2,3].map(funA).map(funB);
[1,2,3].map(x => funB(funA(x)));
```

```javascript
[1,2,3].map(a => a) == [1,2,3];
```

```javascript
Array.prototype.map = function (func) {
  const acc = [];
  for (let i = 0; i < this.length; i++) {
    acc.push(func(this[i]));
  }
  return acc;
};
```

THE FUNCTOR TOWER

APPLY

Functor.map

```haskell
class Functor f => Apply f where
  apply :: f a -> f (a -> b) -> f b

instance Apply [] where
  apply xs []       = []
  apply xs (f : fs) = map f xs ++ apply xs fs
```

```elixir
defimpl Apply, for: List do
  def apply(_,    []), do: []
  def apply(vals, [func | fs]) do
    map(vals, func) ++ apply(vals, fs)
  end
end
```

```javascript
Array.prototype.apply = function (funs) {
  const acc = [];
  for (let i = 0; i < funs.length; i++) {
    const row = this.map(funs[i]);
    acc.push(row);
  }
  return acc;
};
```

JS

THE FUNCTOR TOWER
APPLY

Functor.map
|
Apply.apply

```haskell
class Functor f => Apply f where
  apply :: f a -> f (a -> b) -> f b

instance Apply [] where
  apply xs []       = []
  apply xs (f : fs) = map f xs ++ apply xs fs
```

```elixir
defimpl Apply, for: List do
  def apply(_,    []), do: []
  def apply(vals, [func | fs]) do
    map(vals, func) ++ apply(vals, fs)
  end
end
```

```javascript
Array.prototype.apply = function (funs) {
  const acc = [];
  for (let i = 0; i < funs.length; i++) {
    const row = this.map(funs[i]);
    acc.push(row);
  }
  return acc;
};
```

JS

THE FUNCTOR TOWER
APPLICATIVE

```haskell
class Apply f => Applicative f where
  wrap :: a -> f a

instance Applicative [] where
| wrap x = [x]
```

```elixir
defprotocol Applicative do
  def wrap(proxy, to_wrap)
end

defimpl Applicative, for: List do
  def wrap(_, to_wrap), do: [to_wrap]
end
```

```javascript
Array.prototype.wrap = toWrap => [toWrap];
```

JS

Functor.map
|
Apply.apply

🙋‍♀️ AKA return, pure, of, unit

THE FUNCTOR TOWER
APPLICATIVE

```haskell
class Apply f => Applicative f where
  wrap :: a -> f a

instance Applicative [] where
| wrap x = [x]
```

```elixir
defprotocol Applicative do
  def wrap(proxy, to_wrap)
end

defimpl Applicative, for: List do
  def wrap(_, to_wrap), do: [to_wrap]
end
```

```javascript
Array.prototype.wrap = toWrap => [toWrap];
```

JS

Functor.map

|

Apply.apply

Applicative.wrap

🤦‍♀️ AKA return, pure, of, unit

THE FUNCTOR TOWER

CHAIN

Functor.map

|

Apply.apply

Applicative.wrap

```haskell
class Applicative f => Chain f where
  bind :: f a -> (a -> f b) -> f b

instance Chain [] where
  bind []       _       = []
  bind (x : xs) chainer = chainer x ++ bind xs chainer
```

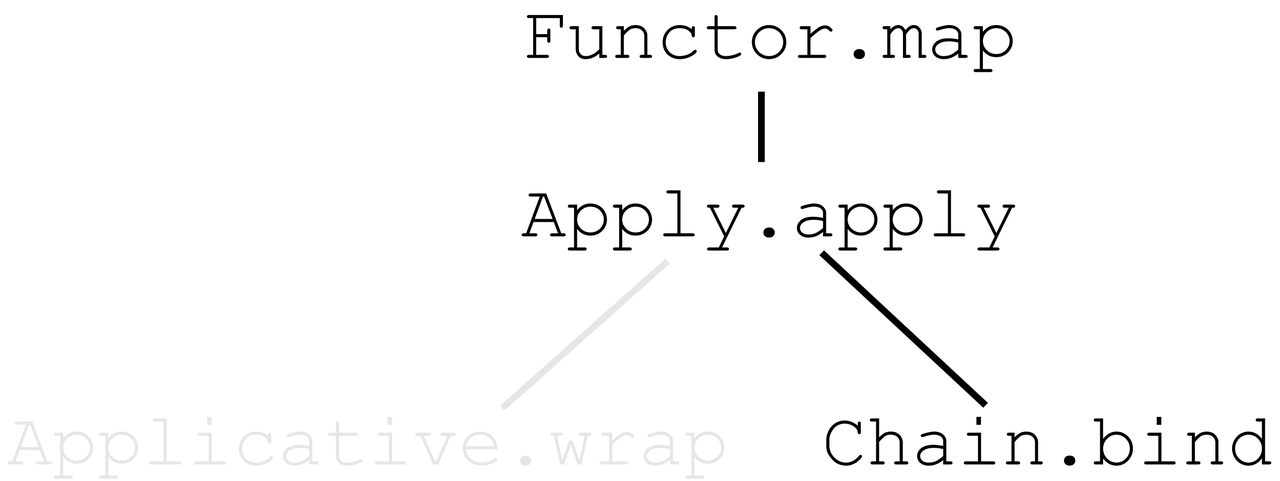
```elixir
defimpl Chain, for: List do
  def bind([], chainer), do: []
  def bind([x | xs], chainer) do
    chainer(x) ++ bind(xs, chainer)
  end
end
```

```javascript
Array.prototype.bind = function (chainer) {
  return Array.flat(this.map(chainer));
}
```

JS

THE FUNCTOR TOWER

CHAIN



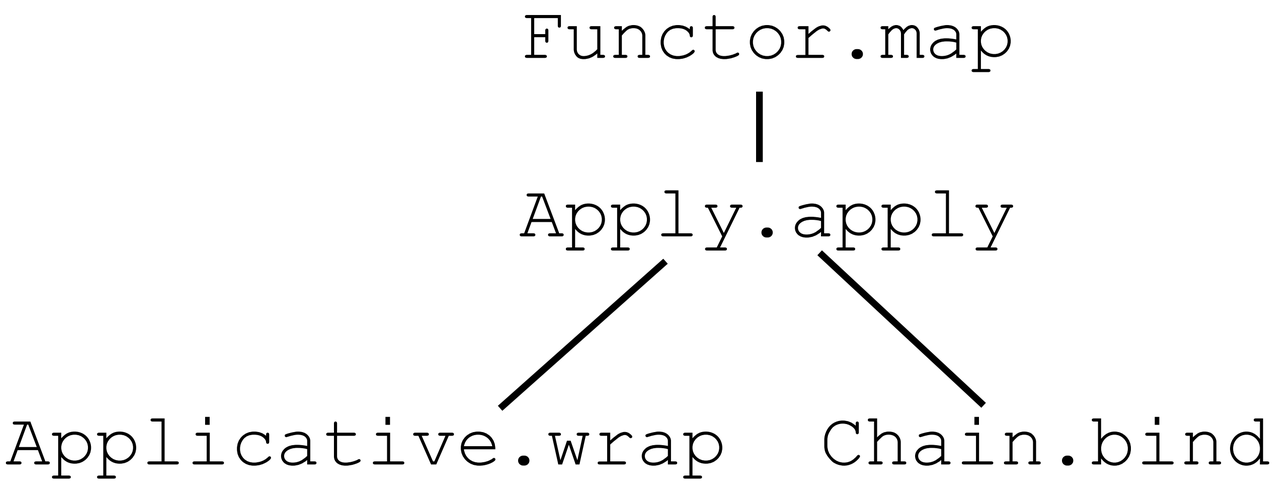

```haskell
class Applicative f => Chain f where
  bind :: f a -> (a -> f b) -> f b

instance Chain [] where
  bind []       _       = []
  bind (x : xs) chainer = chainer x ++ bind xs chainer
```

```elixir
defimpl Chain, for: List do
  def bind([], chainer), do: []
  def bind([x | xs], chainer) do
    chainer(x) ++ bind(xs, chainer)
  end
end
```

```javascript
Array.prototype.bind = function (chainer) {
  return Array.flat(this.map(chainer));
}
```

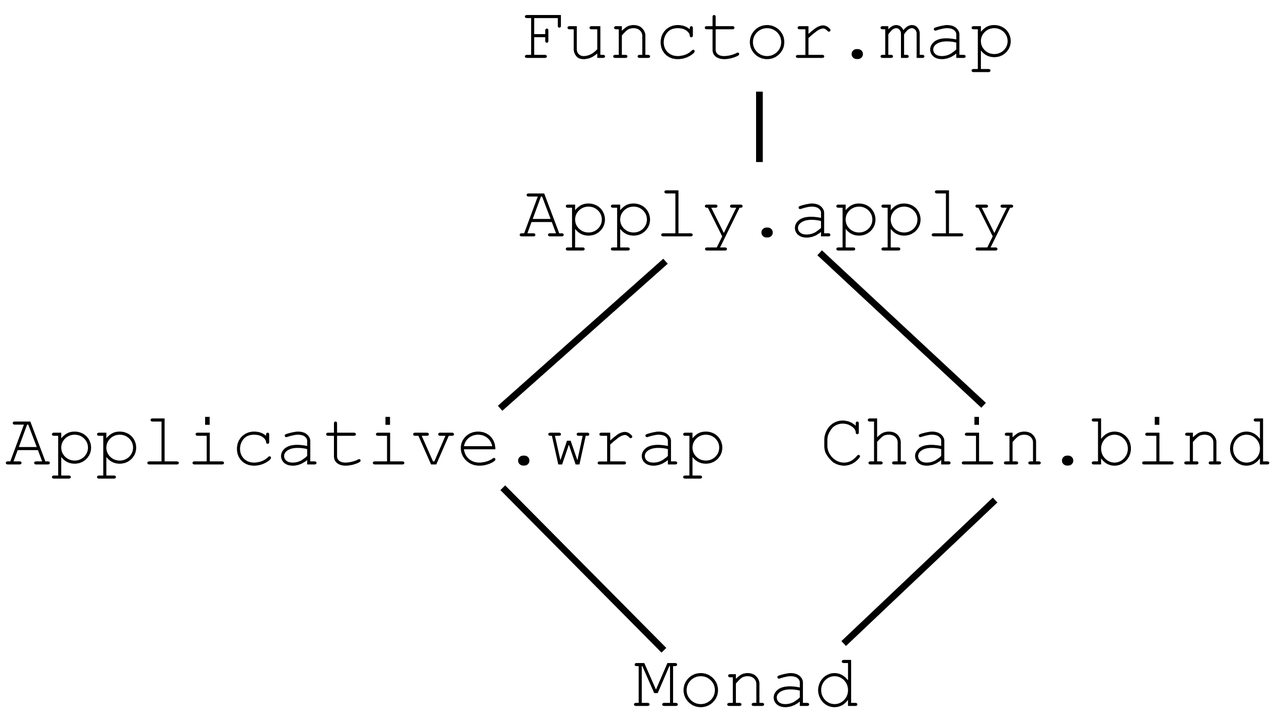
THE FUNCTOR TOWER

MONAD 🦸



THE FUNCTOR TOWER

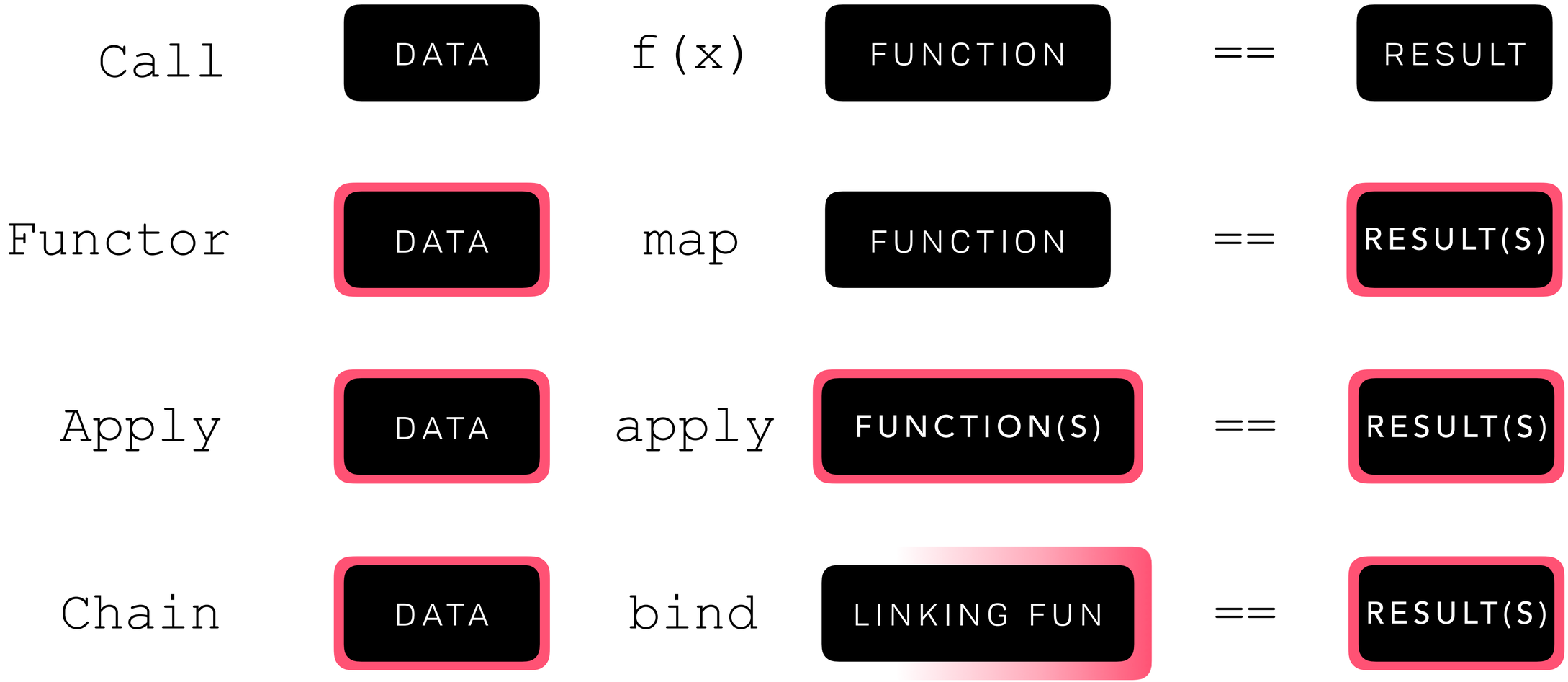
MONAD 🦸



- Monads are the essence of Turing complete, effectual computation

THE FUNCTOR TOWER

NOT GROUPED BY ACCIDENT



THE FUNCTOR TOWER

MONAD (FINALLY!)

STRUCTURED ABSTRACTION

BUT MAYBE NOT SO DIFFERENT

```
const result    = await runProcess();
const nextResult = await doThing(result);
const moreResult = await nextFunc(nextResult);
const final     = await moreFunc(moreResult);
```

```
runProcess
  .then(result => {
    return doThing(result).then(nextResult => {
      return nextFunc(nextResult).then(moreResult => {
        return moreFunc(moreResult);
      });
    });
  });
```

```
try {
  explodingFunc();
} catch(error) {
  handleOrReport(error);
}
```

```
function foo() {
  console.trace();

  function bar() {
    console.trace();
  }

  bar();
}

foo();
```

EITHER/RESULT

EITHER/RESULT

RAILROAD PROGRAMMING

EITHER/RESULT

RAILROAD PROGRAMMING

```
try {
  explodingFunc();
  dangerousFunc();
  badFunc();
  mightFailFunc();
} catch(error) {
  handleOrReport(error);
}
```

EITHER/RESULT

RAILROAD PROGRAMMING

```
try {
  explodingFunc();
  dangerousFunc();
  badFunc();
  mightFailFunc();
} catch(error) {
  handleOrReport(error);
}
```

Happy Path (Continue)

Error Case (Skip)

No Effect (Afterwards)

EITHER/RESULT

RAILROAD PROGRAMMING

```
try {
  explodingFunc();
  dangerousFunc();
  badFunc();
  mightFailFunc();
} catch(error) {
  handleOrReport(error);
}
```

Happy Path (Continue)

Error Case (Skip)

No Effect (Afterwards)

EITHER/RESULT

RAILROAD PROGRAMMING

```
try {
  explodingFunc();
  dangerousFunc();
  badFunc();
  mightFailFunc();
} catch(error) {
  handleOrReport(error);
}
```

Happy Path (Continue)

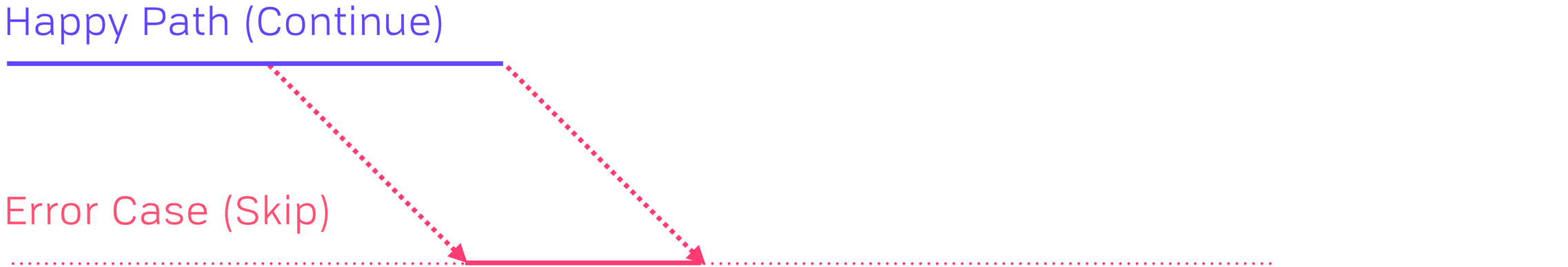
Error Case (Skip)

No Effect (Afterwards)

EITHER/RESULT

RAILROAD PROGRAMMING

```
try {
  explodingFunc();
  dangerousFunc();
  badFunc();
  mightFailFunc();
} catch(error) {
  handleOrReport(error);
}
```

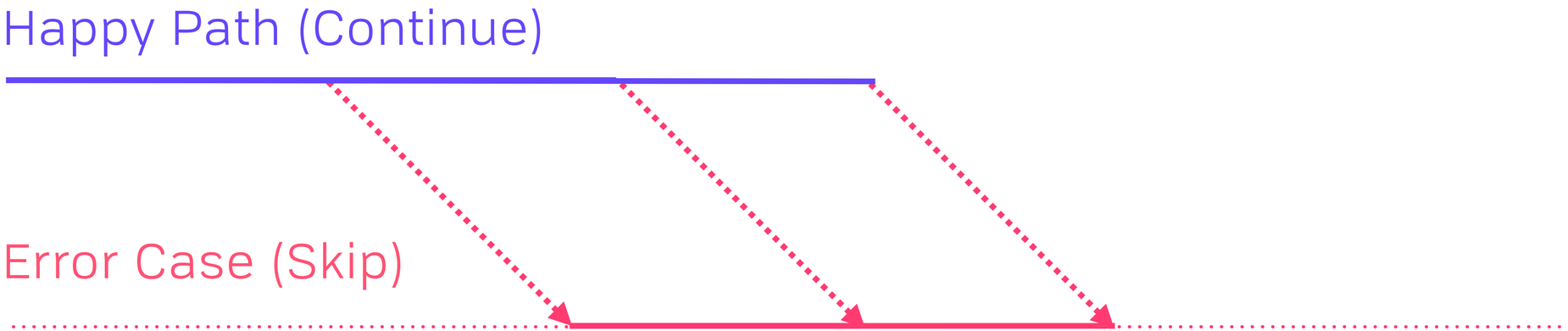


No Effect (Afterwards)

EITHER/RESULT

RAILROAD PROGRAMMING

```
try {
  explodingFunc();
  dangerousFunc();
  badFunc();
  mightFailFunc();
} catch(error) {
  handleOrReport(error);
}
```

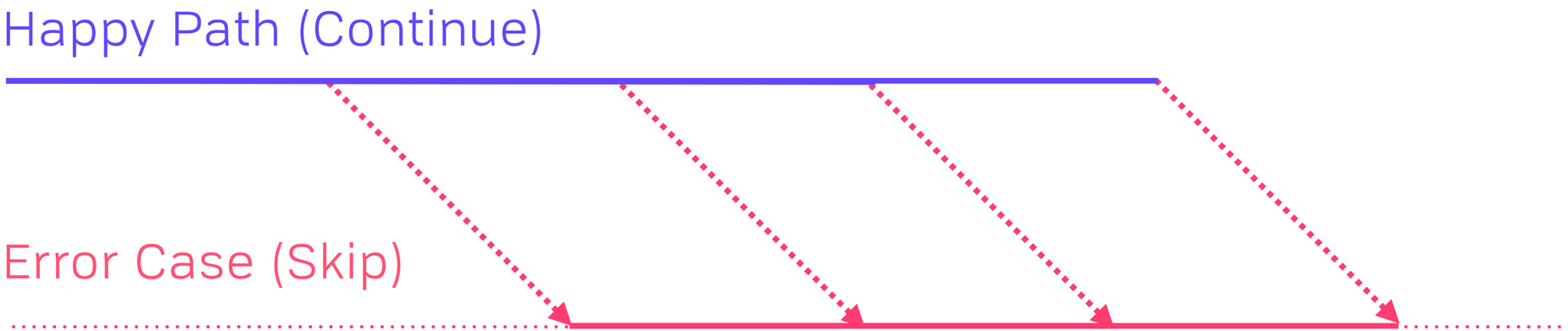


No Effect (Afterwards)

EITHER/RESULT

RAILROAD PROGRAMMING

```
try {
  explodingFunc();
  dangerousFunc();
  badFunc();
  mightFailFunc();
} catch(error) {
  handleOrReport(error);
}
```

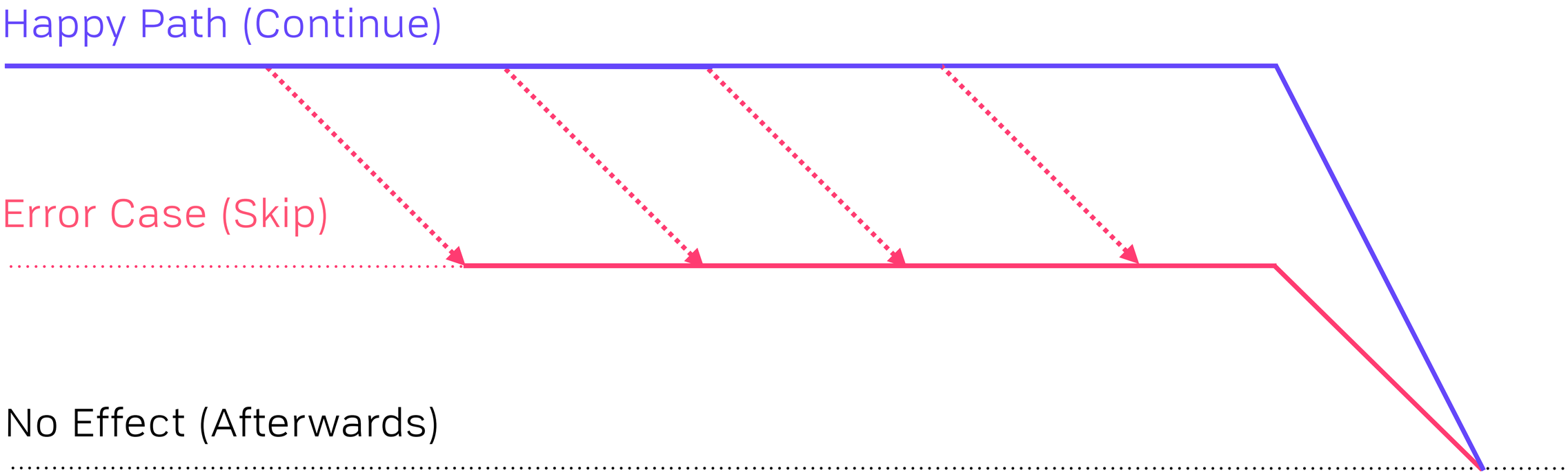


No Effect (Afterwards)

EITHER/RESULT

RAILROAD PROGRAMMING

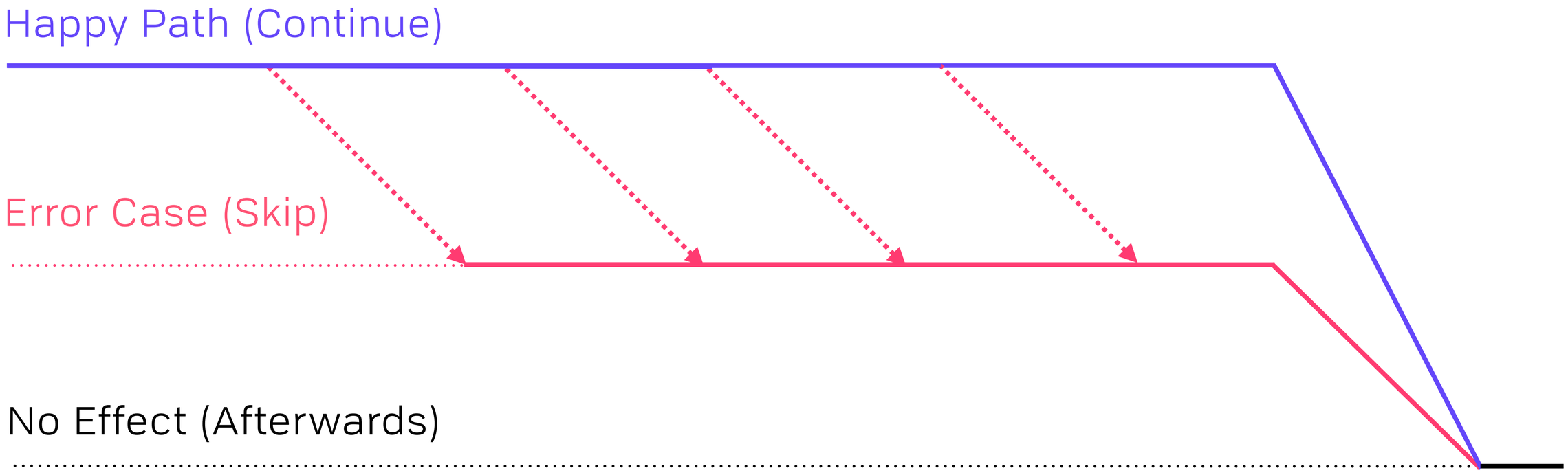
```
try {
  explodingFunc();
  dangerousFunc();
  badFunc();
  mightFailFunc();
} catch(error) {
  handleOrReport(error);
}
```



EITHER/RESULT

RAILROAD PROGRAMMING

```
try {
  explodingFunc();
  dangerousFunc();
  badFunc();
  mightFailFunc();
} catch(error) {
  handleOrReport(error);
}
```



EITHER/RESULT

CARRIER DATA

```haskell
data Result err value
  = Ok value
  | Error err
```

```elixir
defmodule Result do
  @type t :: Error.t() | Ok.t()

  defmodule Error do
    @type t :: %Error{err: any()}
    defstruct :err
  end

  defmodule Ok do
    @type t :: %Ok{value: any()}
    defstruct :value
  end
end
```

```javascript
class Ok {
  constructor(value) { this.value = value; }
```

```javascript
class Error {
  constructor(err) { this.err = err; }
```

JS

EITHER/RESULT
INSTANCES

```haskell
data Result err value
  = Ok value
  | Error err

instance Functor (Result err value) where
  fmap _ (Error err) = Error err
  fmap f (Ok value) = Ok (f value)

instance Applicative (Result err value) where
  wrap value = Ok value

instance Chain (Result err value) where
  bind (Error err) _ = Error err
  bind (Ok value)  f = f value
```

```elixir
defmodule Result do
  @type t :: Error.t() | Ok.t()

  defmodule Error do
    @type t :: %Error{err: any()}
    defstruct :err
  end

  defmodule Ok do
    @type t :: %Ok{value: any()}
    defstruct :value
  end
end

defimpl Functor, for: Error do
  def map(err, _), do: err
end

defimpl Functor, for: Ok do
  def map(%Ok{value: old_value}, fun) do
    %Ok{value: fun.(old_value)}
  end
end
```

```javascript
class Ok {
  constructor(value) { this.value = value; }

  map(fun)    { return new Ok(fun(this.value)); }
  wrap(value) { return new Ok(value); }

  bind(chainer) {
    return new Ok(chainer(this.value));
  }
}

class Error {
  constructor(err) { this.err = err; }

  map(_)      { return this; }
  wrap(value) { return new Ok(value); }
  bind(_)     { return this; }
}
```

JS

EITHER/RESULT
USE

```haskell
evenOrErr :: Int -> Result NotEven Int
evenOrErr num =
  if rem num 2 == 0
     then wrap num
     else Err (NotEven num)

Ok 42 >>= evenOrErr
      >>= \num -> wrap (toString num)
      >>= shortOrErr

do
  let num = 42
  even <- evenOrErr num
  shortOrErr $ toString even

shortOrErr . toString =<< evenOrErr 42
```

```elixir
def even_or_err(num) do
  if rem(num, 2) do
    %Ok{value: num}
  else
    %Error{err: {:odd, num}}
  end
end

%Ok{value: 42}
|> bind(&even_or_err/2)
|> map(&to_string/1)
|> short_or_err()
# => %Ok{value: "42"}
```

```elixir
def return(val), do: wrap(val)
def my_throw(err), do: %Error{err: err}

def even_or_err(num) do
  if rem(num, 2) do
    return(num)
  else
    my_throw({:odd, num})
  end
end
```

```javascript
const evenOrErr = num =>
  num % 2 === 0 ? new Ok(num) : new Error([num, "not even"]);

const start = new Ok(42);

start
| .bind(num  => evenOrErr(num))
| .bind(even => even.toString())
| .bind(str  => shortOrErr(str));
```

EITHER/RESULT

USE

```haskell
evenOrErr :: Int -> Result NotEven Int
evenOrErr num =
  if rem num 2 == 0
     then wrap num
     else Err (NotEven num)

Ok 42 >>= evenOrErr
      >>= \num -> wrap (toString num)
      >>= shortOrErr

do
  let num = 42
  even <- evenOrErr num
  shortOrErr $ toString even

shortOrErr . toString =<< evenOrErr 42
```

```elixir
def even_or_err(num) do
  if rem(num, 2) do
    %Ok{value: num}
  else
    %Error{err: {:odd, num}}
  end
end

%Ok{value: 42}
|> bind(&even_or_err/2)
|> map(&to_string/1)
|> short_or_err()
# => %Ok{value: "42"}
```

```elixir
def return(val), do: wrap(val)
def my_throw(err), do: %Error{err: err}

def even_or_err(num) do
  if rem(num, 2) do
    return(num)
  else
    my_throw({:odd, num})
  end
end
```

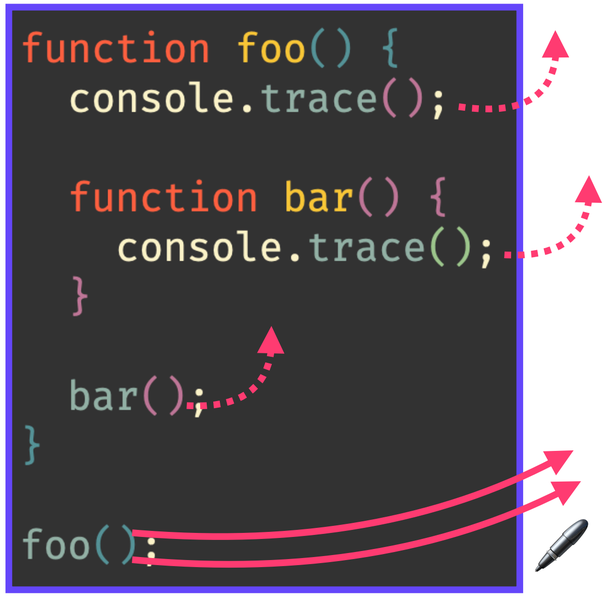
```javascript
const evenOrErr = num =>
  num % 2 === 0 ? new Ok(num) : new Error([num, "not even"]);

const start = new Ok(42);

start
| .bind(num  => evenOrErr(num))
| .bind(even => even.toString())
| .bind(str  => shortOrErr(str));
```

WRITER

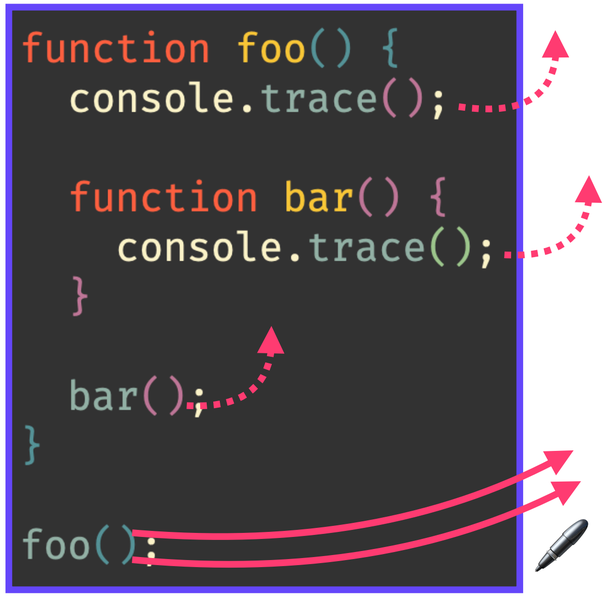
WRITER

INSPECTABLE, DATA-ORIENTED LOGGING

WRITER MONAD



WRITER MONAD

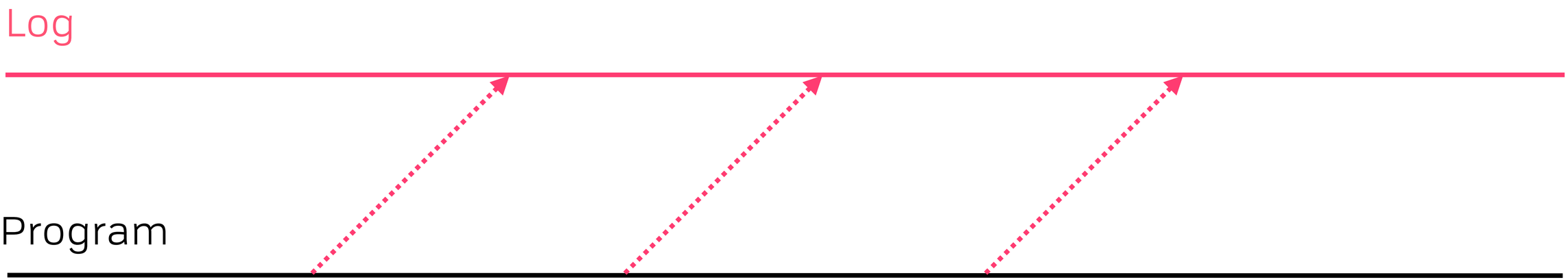


Log

Program

WRITER MONAD

```
function foo() {
  console.trace();

  function bar() {
    console.trace();
  }

  bar();
}

foo();
```



WRITER
CARRIER DATA

```haskell
data Writer value log
  = Writer (value, log)
```

```elixir
defmodule Writer do
  defstruct [:value, :log]
end
```

```javascript
class Writer {
  constructor(value, log = []) {
    this.value = value;
    this.log   = log;
  }
```

WRITER
INSTANCES

```elixir
defimpl Functor, for: Writer do
  def map(%Writer{writer: {value, log}}, fun) do
    %Writer{value: fun.(value), log: log}
  end
end

defimpl Applicative, for: Writer do
  def wrap(%Writer{writer: {_, log}}, value) do
    %Writer{value: value, log: empty(log)}
end

defimpl Chain, for: Writer do
  def chain(%Writer{value: old_value, log: old_log}, chainer) do
    %Writer{
      value: new_value,
      log:   new_log
    } = chainer.(old_value)

    %Writer{value: new_value, log: old_log <> new_log}
  end
end
```

```haskell
instance Functor (Writer val log) where
  map f (Writer log val) = Writer ((f val),  log)

instance Monoid log => Applicative (Writer log val) where
  wrap val = Writer (value, empty)

instance Semigroup log => Chain (Writer log val) where
  bind (Writer (oldVal, oldLog)) f =
    Writer (newVal, concat oldLog newLog)
    where
      Writer (newVal, newLog) = f oldVal
```

```javascript
class Writer {
  constructor(value, log = []) {
    this.value = value;
    this.log   = log;
  }

  map(func) {
    this.value = func(this.value);
  }

  bind(chainer) {
    const newWriter = chainer(this.value);
    this.value = newWriter.value;
    this.log   = this.log.concat(newWriter.log);
  }
}
```

JS

WRITER
USE

```haskell
tell log = wrap ((), log)

half num = num / 2 >>= \half ->
  wrap (half, [show num ++ " / 2 = " ++ show half])
```

```elixir
def half(num) do
  half = num / 2

  %Writer{
    value: half,
    log:   ["#{num} / 2 = #{half}"]
  }
end
```

```javascript
const tell = log => new Writer(null, log);

const half = num => {
  const halved = num / 2;
  const writer = new Writer(halved);
  writer
    .tell([`#{num} / 2 = #{half}`])
    .bind(_ => new Writer(halved));
};
```

```haskell
half 42 >>= half >>= half

{-
  Writer ( 5.25
         , [ "42 / 2 = 21.0"
           , "21.0 / 2 = 10.5"
           , "10.5 / 2 = 5.25"
           ]
         )
-}
```

```elixir
42
|> half()
|> bind(&half/1)
|> bind(&half/1)

%Writer{
  value: 5.25,
  log: [
    "42 / 2 = 21.0",
    "21.0 / 2 = 10.5",
    "10.5 / 2 = 5.25"
  ]
}
```

```javascript
const logged =
  half(42)
    .bind(half)
    .bind(half);

logged.value; //=> 5.25
logged.log;
/* => [
    "42 / 2 = 21.0",
    "21.0 / 2 = 10.5",
    "10.5 / 2 = 5.25"
*/
```

READER

READER

CONFIG OR CONTEXT INJECTION

READER
MONAD

READER
MONAD

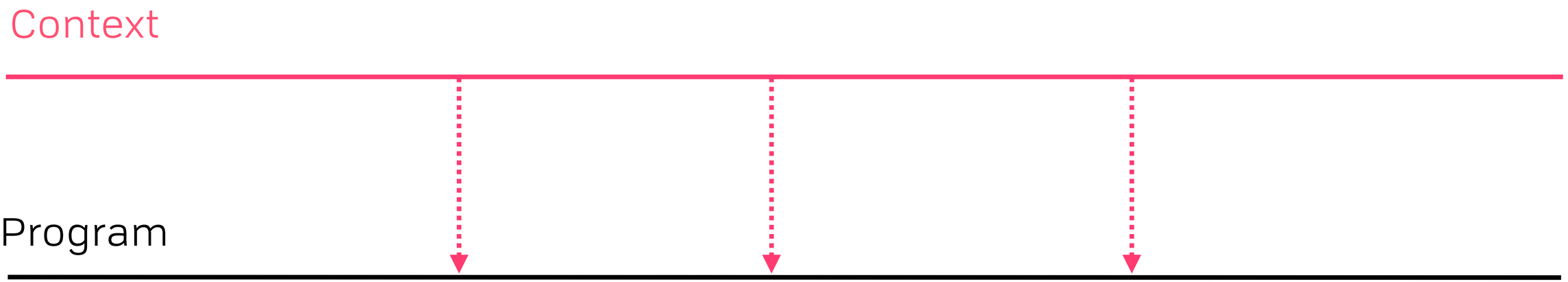
```
export const themes = {
  // theme data
};

export const ThemeContext = React.createContext(
  themes.dark // default value
);

// ...

let theme = this.context;
```

Context

Program

READER MONAD

```
export const themes = {
  // theme data
};

export const ThemeContext = React.createContext(
  themes.dark // default value
);

// ...

let theme = this.context;
```



READER

CARRIER DATA

```haskell
newtype Reader env a
  = Reader { runReader :: env -> a }
```

```elixir
defmodule Reader do
  @type t :: %Reader{reader: fun()}
  struct :reader
end
```

```javascript
class Reader {
  constructor(reader) {
    this.reader = reader;
  }
}
```

JS

READER
INSTANCES

```elixir
defmodule Reader do
  @type t :: %Reader{reader: fun()}
  struct :reader

  def run(%Reader{reader: fun}, arg), do: fun.(arg)
end
```

```elixir
defimpl Functor, for: Reader do
  def map(%Reader{reader: inner}, fun) do
    Reader.new(fn env -> env |> inner.() |> fun.() end)
  end
end

defimpl Applicative, for: Reader do
  def wrap(_, value) do
    Reader.new(fn _ -> value end)
  end
end

defimpl Chain, for: Reader do
  def bind(reader, chainer) do
    %Reader{reader: fn env ->
      reader
      |> Reader.run(env)
      |> chainer.()
      |> Reader.run(env)
    end}
  end
end
```

```haskell
newtype Reader env a
  = Reader { runReader :: env -> a }

instance Functor (Reader env a) where
  map f (Reader inner) =
    Reader (\env -> f (inner env))

instance Applicative (Reader env a) where
  wrap a = Reader (\e -> a)

instance Chain (Reader env a) where
  bind (Reader inner) f =
    Reader (\env -> runReader (f (inner env)) env)
```

```javascript
class Reader {
  constructor(reader) {
    this.reader = reader;
  }

  static wrap(value) {
    return new Reader(env => value);
  }

  run(env) { return this.reader(env); }

  map(fun) {
    return new Reader(env => fun(this.reader(env)));
  }

  bind(chainer) {
    return new Reader(env => {
      Reader.run(chainer(this.reader))(env);
    });
  }
}
```

READER
USE

```elixir
def run(%Reader{reader: fun}, env), do: fun.(env)
def ask(), do: Reader{reader: fn x -> x end}

def my_fun() do
  ask()
  |> bind(fn env -> Enum.count(env) end)
  |> bind(fn count ->
    "The env has #{count} elements"
  end)
end

run(&myFun/0).([1,2,3])     # "The env has 3 elements"
run(&myFun/0).([1,2,3,4,5]) # "The env has 3 elements"
```

```haskell
ask = Reader (\env -> env)

myFun =
  ask >>= \env ->
    length env >>= count ->
      "The env has " ++ show count ++ "elements"

runReader [1,2,3]     myFun -- "The env has 3 elements"
runReader [1,2,3,4,5] myFun -- "The env has 5 elements"
```

```javascript
const ask = new Reader(a => a);

const myFun =
  ask.bind(env => env.length)
     .bind(count => `The env has ${count} elements`);

myFun.run([1,2,3]);     // The env has 3 elements
myFun.run([1,2,3,4,5]); // The env has 5 elements
```

MONADS

MONADS

A LOT OF POWER FOR A HANDFUL OF FUNCTIONS

https://fission.codes
https://tools.fission.codes

🇸🇪 THANK YOU, MALMÖ 🎉

brooklyn@fission.codes
github.com/expede
@expede