DECONSTRUCTING

Lambdas

AN awkward GUIDE TO PROGRAMMING WITHOUT functions

CHRIS PENNER

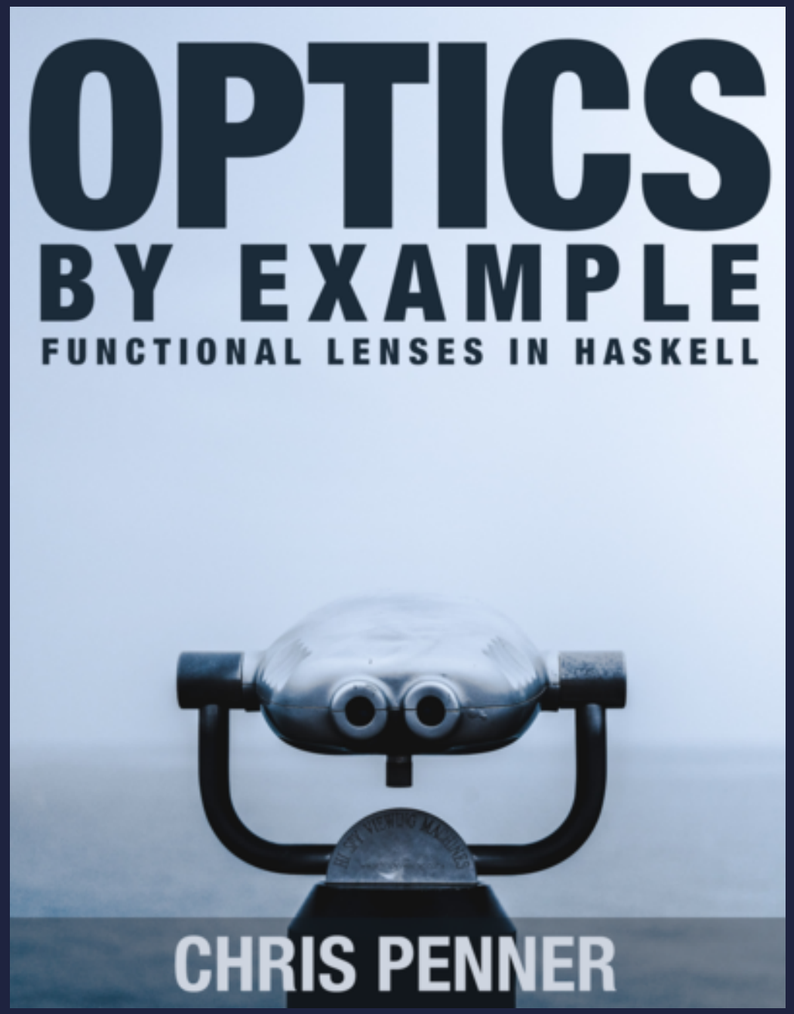
I like doing handstands

I write Haskell

I wrote Optics By Example

chrispenner.ca

@ChrisLPenner



Coupon code:
leanpub.com/optics-by-example/c/berlin

What are functions *really?*
Functions considered *harmful*
Rebuilding functions from *first principles*
Functions as *data*

ACT I.

WHAT'S THE DEAL WITH

Functions

Why are they so

FOUNDATIONAL

IN MODERN

PROGRAMMING LANGUAGES?

Incredibly

ADAPTABLE

ENCAPSULATION

ABSTRACTION

STACKABLE

COMPOSABLE

It's reasonable to think of your program as
one big function

Modelled on the Lambda Calculus
which is Turing complete

Clearly very useful

BUT FUNCTIONS HAVE A

DARK SIDE

THEY

assume

WE PLAN TO

execute

THEM

WHAT ELSE WOULD YOU DO?

Analyze

THEM

Show THEM

Debug

THEM

Transpile

THEM

Serialize

THEM

SECURITY

"It's not safe to run arbitrary functions"

ONE OF THE FEW

COMPLETELY OPAQUE

TYPES IN HASKELL

DID WE GET IT RIGHT?

ACT II.

FUNCTIONS ARE

too powerful

Why?

One large powerful type
provides less information
than several small ones

MONADS ARE SIMILAR

We have a primitive monad:

IO

The only thing we can do is *execute* it

What does this thing do?

```
doTheThing :: IO ()
```

```
doTheThing :: IO ()
```

```
doTheThing :: IO ()
```

- Launch the nukes

```
doTheThing :: IO ()
```

- Launch the nukes
- Heat up your CPU

```
doTheThing :: IO ()
```

- Launch the nukes
- Heat up your CPU
- Load a picture of a cute cat

By breaking it down we can discover what it's doing

Constraints provide insight

```
doTheThing ::
  ( MonadLogger m
  , MonadDB m
  , MonadState User m
  ) => m ()
```

Rule of least power

The more powerful the tool

The less we can know about how it's used.

Functions are similar

Functions have
too many behaviours
bundled together

We lose the ability to

introspect

I can't tell whether a function

loops or recurses

I can't tell whether a function
is *total*

I can't tell a function's
execution complexity

I can't reinterpret its behaviours

into Javascript

All we can do is

execute

We need to identify and abstract over each independent behaviour

ACT III.

Functions

from

SCRATCH

THE *Arrow* TYPECLASS

IS ONE ATTEMPT

```
class Category a => Arrow a where
    -- | Lift a function to an arrow.
    arr :: (b -> c) -> a b c
    ...
```

Oops... Back to square one!

Let's try again...

First things first.

What can a *Function* actually do?

By defining classes
for our behaviours
we can choose any interpretation
which implements them

LET'S LOOK AT SOME BEHAVIOURS

FUNCTIONS

Compose

```
g . f
```

CATEGORY

```
class Category k where
  id    :: k x x

  (.) :: k y z
      -> k x y
      -> k x z
```

CATEGORY

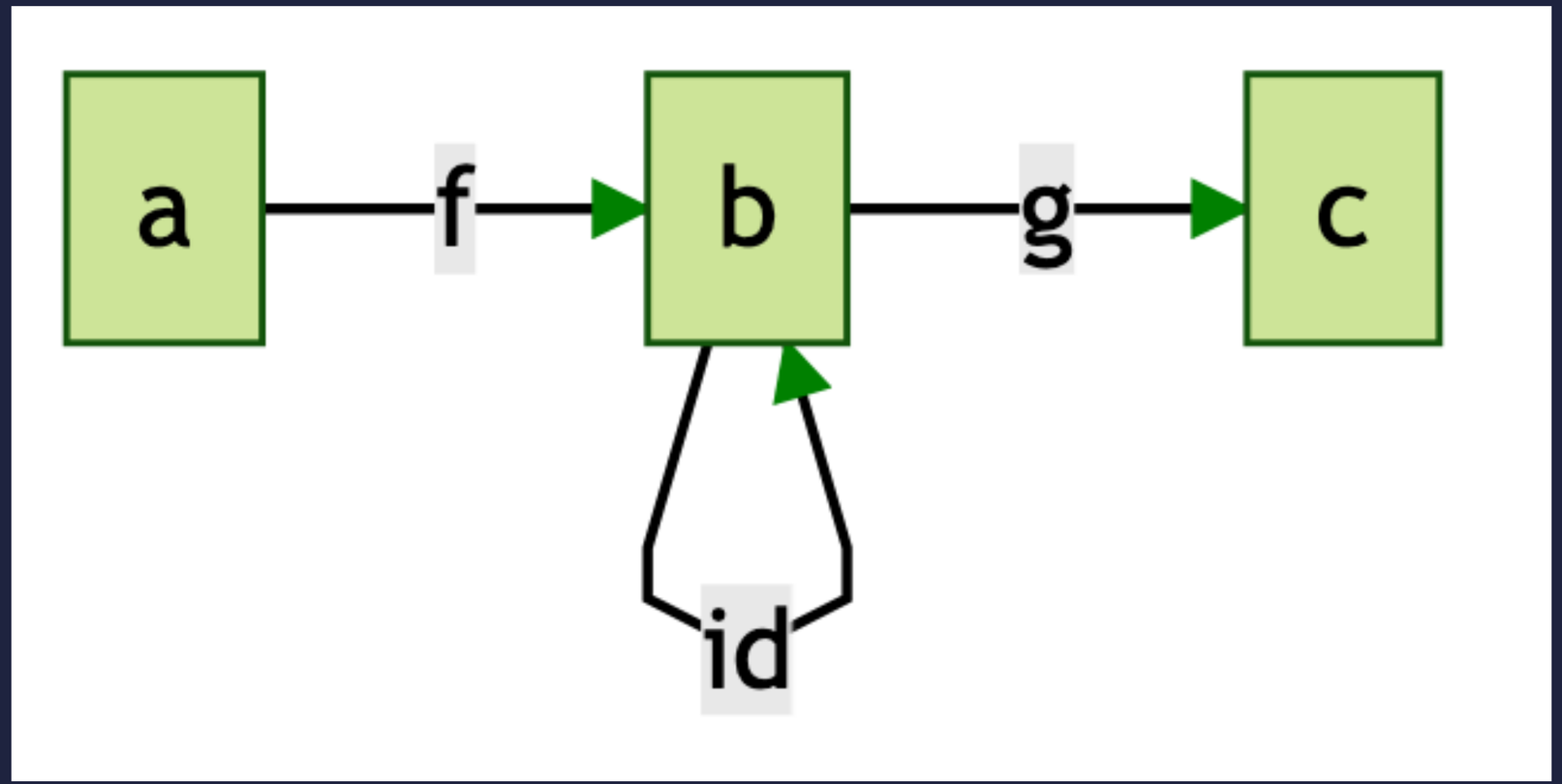
```
class Category k where
  id    :: k x x

  (.) :: k y z
      -> k x y
      -> k x z
```

We'll mostly use

```
(>>>) :: k x y
      -> k y z
      -> k x z
```

CATEGORY

```
f >>> id >>> g
```



We can already do interesting things!

We can define combinators using only constraints

```
thrice :: Category k => k a a -> k a a
thrice k = k >>> k >>> k
```

STILL WORKS WITH FUNCTIONS

```
>>> add3 = thrice (+1)
>>> add3 10
13
```

STILL WORKS WITH FUNCTIONS

```
>>> add3 = thrice (+1)
>>> add3 10
13
```

NEW CATEGORY

Javascript Functions

JAVASCRIPT CATEGORY

Javascript values form the Objects
Javascript functions form the Arrows

JAVASCRIPT CATEGORY

Identity: The identity function in JS

Composition: Composition of javascript functions

```haskell
newtype JSFunc a b =
  JSFunc {renderJS :: Text}
```

The Phantom Types keep things safe

LET'S WRITE OUR FIRST INTERPRETER

```
instance Category JSFunc where
  id :: JSFunc x x
  id = JSFunc "(x => x)"
  (>>>) :: JSFunc x y -> JSFunc y z -> JSFunc x z
  f >>> g = JSFunc [i|(input) => {
  const fst = #{renderJS f};
  const snd = #{renderJS g};
  return snd(fst(input));
}|]
```

```
instance Category JSFunc where
  id :: JSFunc x x
  id = JSFunc "(x => x)"
  (>>>) :: JSFunc x y -> JSFunc y z -> JSFunc x z
  f >>> g = JSFunc [i|(input) => {
  const fst = #{renderJS f};
  const snd = #{renderJS g};
  return snd(fst(input));
}|]
```

```
times10 :: JSFunc Int Int
times10 = JSFunc "(x => x * 10)"

times1000 :: JSFunc Int Int
times1000 = thrice times10
```

```
times10 :: JSFunc Int Int
times10 = JSFunc "(x => x * 10)"

times1000 :: JSFunc Int Int
times1000 = thrice times10
```

```
>>> renderJS times1000
(input) => {
  const fst = (x => x * 10);
  const snd = (input) => {
    const fst = (x => x * 10);
    const snd = (x => x * 10);
    return snd(fst(input));
  };
  return snd(fst(input));
}
```

```
> times1000 = (input) => {
    const fst = (x => x * 10);
    const snd = (input) => {
      const fst = (x => x * 10);
      const snd = (x => x * 10);
      return snd(fst(input));
    };
    return snd(fst(input));
  }
<· (input) => {
    const fst = (x => x * 10);
    const snd = (input) => {
      const fst = (x => x * 10);
      const snd = (x => x * 10);
      return snd(fst(input));
    };
    return snd(fst(input));
  }
> times1000(32)
<· 32000
>
```

We just wrote a Javascript function in *Haskell*
without any compiler hacking!

Let's add some more behaviours

Working in Haskell
Means a lot of assumptions

In *Haskell*,
Functions can copy or discard their arguments at will

Other categories can't!

Linear functions

C

Rust

Electrical Circuitry

```haskell
class Category k => Cartesian k where
  copy    :: k a (a, a)
  consume :: k a ()
  fst'    :: k (l, r) l
  snd'    :: k (l, r) r
```

```
class Category k => Cartesian k where
  copy    :: k a (a, a)
  consume :: k a ()
  fst'    :: k (l, r) l
  snd'    :: k (l, r) r
```

```
class Category k => Cartesian k where
  copy    :: k a (a, a)
  consume :: k a ()
  fst'    :: k (l, r) l
  snd'    :: k (l, r) r
```

```
class Category k => Cartesian k where
  copy    :: k a (a, a)
  consume :: k a ()
  fst'    :: k (l, r) l
  snd'    :: k (l, r) r
```

```
class Category k => Cartesian k where
  copy    :: k a (a, a)
  consume :: k a ()
  fst'    :: k (l, r) l
  snd'    :: k (l, r) r
```

```
instance Cartesian JSFunc where
  copy    = JSFunc "(x => ([x, x]))"
  consume = JSFunc "(x => null)"
  fst'    = JSFunc "(([x, _]) => x)"
  snd'    = JSFunc "(([_, y]) => y)"
```

Plumbing

Functions can route their arguments
to other internal functions

WHAT'S ALL GOING ON HERE?

```
isPalindrome :: String -> Bool
isPalindrome str =
  str == reverse str
```

How do we do this without applying functions?

```
isPalindrome :: String -> Bool
isPalindrome =
      (\str -> (str, str))  -- copy
  >>> (\(s, s') -> (reverse s, s')) -- ???
  >>> (\(revS, s) -> revS == s)
```

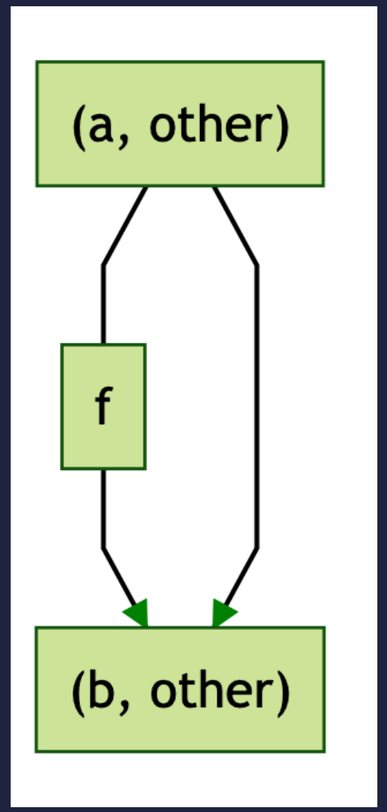
STRONG[1]

```
class Category k => Strong k where
  first' :: k a b
         -> k (a, other) (b, other)

  second' :: k a b
          -> k (other, a) (other, b)
```

[1] some classes adapted from Ed Kmett's `profunctors` library

STRONG

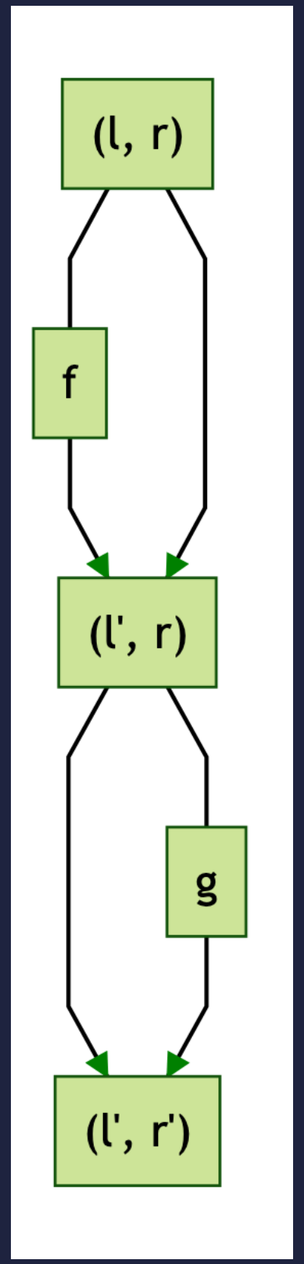
```
first' (f :: k a b)
```

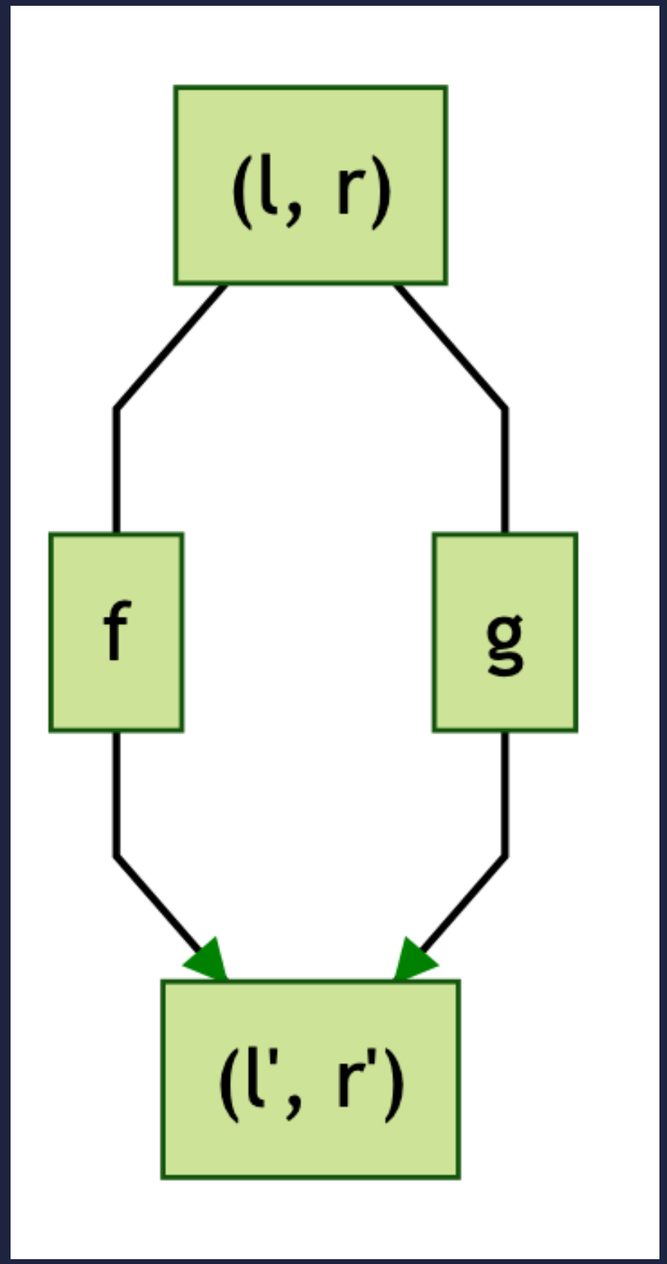


Now we can start building

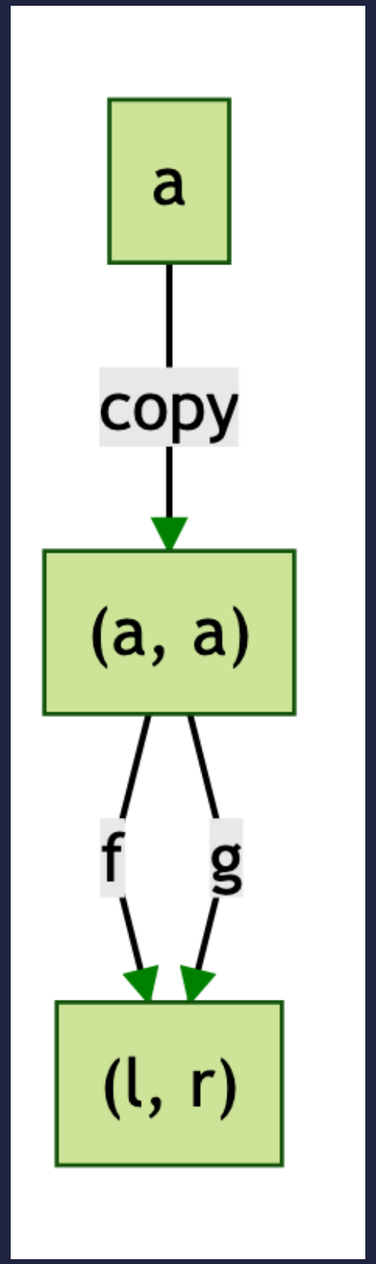
COMBINATORS

```
(***) :: (Category k, Strong k)
      => k l l'
      -> k r r'
      -> k (l, r) (l', r')
(***) l r = first' l >>> second' r
```


(l, r)
f
(l', r)
g
(l', r')


(l, r)
f
g
(l', r')

```
(&&&) :: (Cartesian k, Strong k)
      => k a l
      -> k a r
      -> k a (l, r)
(&&&) l r = copy >>> (l *** r)
```


a
copy
(a, a)
f
g
(l, r)

JAVASCRIPT FUNCTIONS ARE STRONG

```
instance Strong JSFunc where
  first' f = JSFunc [i|([l, r]) => {
  const onFirst = #{renderJS f};
  const result = onFirst(l);
  return [result, r];
}|]
```

LET'S BUILD SOMETHING!

```
isPalindrome :: String -> Bool
isPalindrome str =
  str == reverse str
```

```haskell
isPalindrome :: String -> Bool
isPalindrome =
      (\str -> (str, str)) -- copy
  >>> (\(s, s') -> (reverse s, s')) -- first' reverse
  >>> (\(revS, s) -> revS == s)
```

```
isPalindrome :: Category k
             => k String Bool
isPalindrome =
      (\str -> (str, str))  -- copy
  >>> (\(s, s') -> (reverse s, s')) -- first'
  >>> (\(revS, s) -> revS == s)
```

```
isPalindrome :: Cartesian k
             => k String Bool
isPalindrome =
      copy
  >>> (\(s, s') -> (reverse s, s')) -- first'
  >>> (\(revS, s) -> revS == s)
```

```
isPalindrome :: (Cartesian k, Strong k)
             => k String Bool
isPalindrome =
      copy
  >>> first' reverse
  >>> (\(revS, s) -> revS == s)
```

Let's talk about

PRIMITIVES

DEFINING PRIMITIVES

```
reverseString :: k String String

eq :: Eq a => k (a, a) Bool
```

Primitives are any operations that we've decided we don't want to break down

```
class MyPrimitives k where
  reverseString :: k String String
  eq            :: Eq a => k (a, a) Bool
```

```haskell
instance MyPrimitives (->) where
  reverseString = reverse
  eq            = uncurry (==)
```

```
instance MyPrimitives JSFunc where
  reverseString
    = JSFunc  [i|((s) => s.split("").reverse().join(''))|]
  eq
    = JSFunc  [i|(([x, y]) => x === y)|]
```

```
isPalindrome :: (MyPrimitives k, Cartesian k, Strong k)
             => k String Bool
isPalindrome =
        copy
    >>> first' reverseString
    >>> eq
```

DOESN'T USE ANY

Functions

```
isPalindrome :: (MyPrimitives k, Cartesian k, Strong k)
             => k String Bool


>>> :t isPalindrome @(->)
isPalindrome @(->) :: String -> Bool

-- Inference is smart enough without the annotation
>>> isPalindrome "evilolive"
True

>>> isPalindrome "turkey"
False
```

```
>>> :t isPalindrome @JSFunc
isPalindrome @JSFunc :: JSFunc String Bool

>>> T.putStrLn $ renderJS isPalindrome
(input) => {
  const fst = ((x) => ([x, x]));
  const snd = (input) => {
  const fst = ([l, r]) => {
  const inner = ((s) => s.split("").reverse().join(''));
  const result = inner(l);
  return [result, r];
};
  const snd = (([x, y]) => x === y);
  return snd(fst(input));
};
  return snd(fst(input));
}
```

```
>>> :t isPalindrome @JSFunc
isPalindrome @JSFunc :: JSFunc String Bool

>>> T.putStrLn $ renderJS isPalindrome
(input) => {
  const fst = ((x) => ([x, x]));
  const snd = (input) => {
  const fst = ([l, r]) => {
  const inner = ((s) => s.split("").reverse().join(''));
  const result = inner(l);
  return [result, r];
};
  const snd = (([x, y]) => x === y);
  return snd(fst(input));
};
  return snd(fst(input));
}
```

```
>>> :t isPalindrome @JSFunc
isPalindrome @JSFunc :: JSFunc String Bool

>>> T.putStrLn $ renderJS isPalindrome
(input) => {
  const fst = ((x) => ([x, x]));
  const snd = (input) => {
  const fst = ([l, r]) => {
  const inner = ((s) => s.split("").reverse().join(''));
  const result = inner(l);
  return [result, r];
};
  const snd = (([x, y]) => x === y);
  return snd(fst(input));
};
  return snd(fst(input));
}
```

```
> const isPalindrome = (input) => {
    const fst = ((x) => ([x, x]));
    const snd = (input) => {
    const fst = ([l, r]) => {
    const inner = ((s) => s.split("").reverse().join(''));
    const result = inner(l);
    return [result, r];
  };
    const snd = (([x, y]) => x === y);
    return snd(fst(input));
  };
    return snd(fst(input));
  }
<- undefined
> isPalindrome("evilolive")
<- true
> isPalindrome("turkey")
<- false
>
```

Questions?

FUNCTIONS

Branch

```haskell
collatzStep :: Int -> Int
collatzStep n =
  if even n then n `div` 2
            else (3 * n) + 1
```

Don't take branching for granted!

It's useful to know *whether* something branches

Branching isn't always trivial

First we need to handle

Coproducts

```haskell
class Category k => Cocartesian k where
  injectL :: k a (Either a b)
  injectR :: k a (Either b a)
  unify :: k (Either a a) a
  -- | tags 'Right' when 'True', 'Left' when 'False'
  tag :: k (Bool, a) (Either a a)
```

```haskell
class Category k => Cocartesian k where
  injectL :: k a (Either a b)
  injectR :: k a (Either b a)
  unify :: k (Either a a) a
  -- | tags 'Right' when 'True', 'Left' when 'False'
  tag :: k (Bool, a) (Either a a)
```

```haskell
class Category k => Cocartesian k where
  injectL :: k a (Either a b)
  injectR :: k a (Either b a)
  unify :: k (Either a a) a
  -- | tags 'Right' when 'True', 'Left' when 'False'
  tag :: k (Bool, a) (Either a a)
```

```
instance Cocartesian JSFunc where
  injectL = JSFunc [i|(x => ({tag: 'left', value: x}))|]
  injectR = JSFunc [i|(x => ({tag: 'right', value: x}))|]
  unify   = JSFunc [i|(x => (x.value))|]
  tag     = JSFunc [i|(([b, x]) => ({tag: b ? 'right' : 'left', value: x}))|]
```

NOW WE'RE READY TO MAKE CHOICES

CHOICE[1]

```
class Category k => Choice k where
  left' :: k a b
        -> k (Either a other) (Either b other)

  right' :: k a b
         -> k (Either other a) (Either other b)
```

[1] some classes adapted from Ed Kmett's `profunctors` library

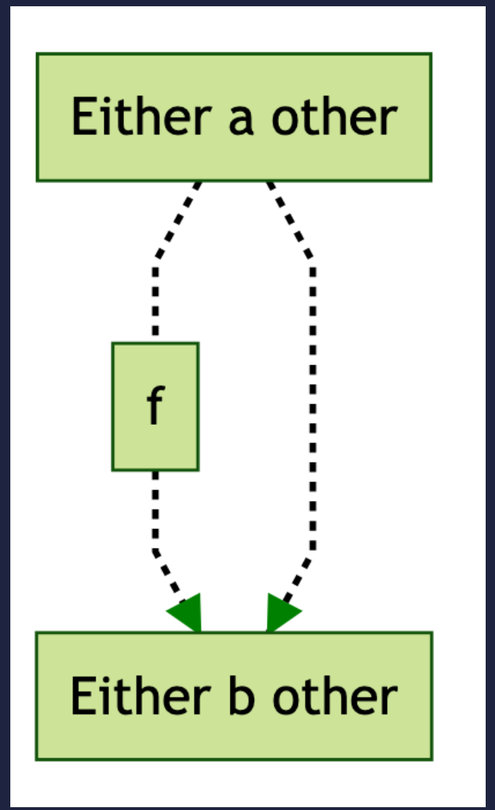
CHOICE[1]

```
class Category k => Choice k where
  left' :: k a b
        -> k (Either a other) (Either b other)

  right' :: k a b
         -> k (Either other a) (Either other b)
```

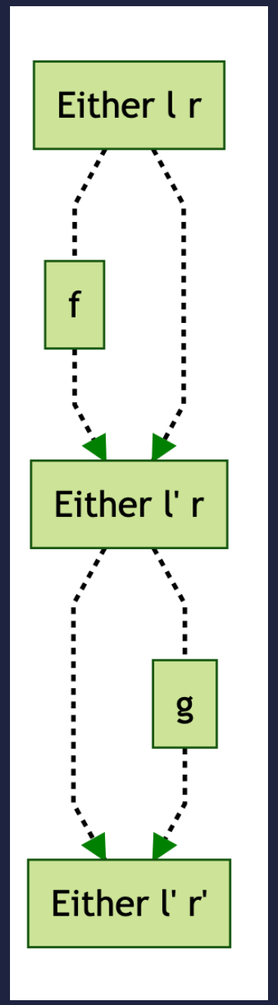
[1] some classes adapted from Ed Kmett's `profunctors` library

CHOICE

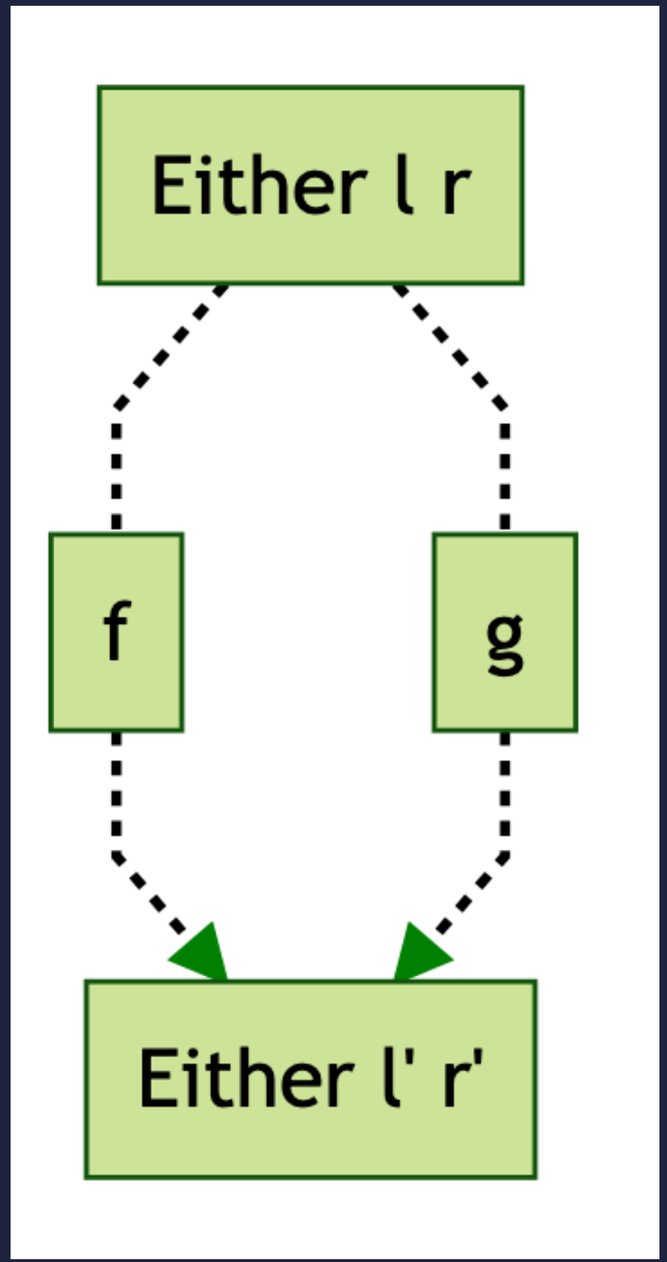
```
left' (f :: k a b)
```



COMBINING WITH CATEGORY



```
(+++) :: (Category k, Choice k)
      => k l l'
      -> k r r'
      -> k (Either l r) (Either l' r')
```


Either l r
f
g
Either l' r'

```
instance Choice JSFunc where
  left :: JSFunc a b
       -> JSFunc (Either a other) (Either b other)

  left f = JSFunc [i|(input) => {
  const overLeft = #{renderJS f};
  if (input.tag == 'left') {
    return { tag: 'left', value: overLeft(input.value) };
  }
  return input
}|]
```

```
instance Choice JSFunc where
  left :: JSFunc a b
       -> JSFunc (Either a other) (Either b other)

  left f = JSFunc [i|(input) => {
  const overLeft = #{renderJS f};
  if (input.tag == 'left') {
    return { tag: 'left', value: overLeft(input.value) };
  }
  return input
}|]
```

```
instance Choice JSFunc where
  left :: JSFunc a b
       -> JSFunc (Either a other) (Either b other)

  left f = JSFunc [i|(input) => {
  const overLeft = #{renderJS f};
  if (input.tag == 'left') {
    return { tag: 'left', value: overLeft(input.value) };
  }
  return input
}|]
```

```
instance Choice JSFunc where
  left :: JSFunc a b
       -> JSFunc (Either a other) (Either b other)

  left f = JSFunc [i|(input) => {
  const overLeft = #{renderJS f};
  if (input.tag == 'left') {
    return { tag: 'left', value: overLeft(input.value) };
  }
  return input
}|]
```

REVISITING COLLATZ

```haskell
collatzStep :: Int -> Int
collatzStep n =
  if even n then n `div` 2
            else (3 * n) + 1
```

Need to detect whether input is even or odd
Then either divide by 2
or multiply by 3 and add 1

Need to handle numbers

```
class Numeric k where
  num     :: Int -> k a Int
  negate' :: k Int Int
  add     :: k (Int, Int) Int
  mult    :: k (Int, Int) Int
  div'    :: k (Int, Int) Int
  mod'    :: k (Int, Int) Int
```

```haskell
instance Numeric (->) where
  num = const
  negate' = negate
  add = uncurry (+)
  mult = uncurry (*)
  div' = uncurry div
  mod' = uncurry mod
```

```
instance Numeric JSFunc where
  num n = JSFunc [i|(x) => (#{n})|]
  negate' = JSFunc [i|(x) => (-x)|]
  add = JSFunc [i|([x, y]) => (x + y)|]
  mult = JSFunc [i|([x, y]) => (x * y)|]
  div' = JSFunc [i|([x, y]) => (x / y)|]
  mod' = JSFunc [i|([x, y]) => (x % y)|]
```

Here's where things get fun!

```haskell
strong :: (Cartesian k, Strong k)
       => k (a, b) r -> k a b -> k a r
strong f x = copy >>> second' x >>> f
```

```haskell
isEven :: forall k. (Numeric k, Cartesian k, Strong k, Choice k)
       => k Int Bool
isEven = mod2 >>> strong eq (num 0)
  where
    mod2 :: k Int Int
    mod2 = strong mod' (num 2)
```

```haskell
isEven :: forall k. (Numeric k, Cartesian k, Strong k, Choice k)
       => k Int Bool
isEven = mod2 >>> strong eq (num 0)
  where
    mod2 :: k Int Int
    mod2 = strong mod' (num 2)
```

```haskell
isEven :: forall k. (Numeric k, Cartesian k, Strong k, Choice k)
       => k Int Bool
isEven = mod2 >>> strong eq (num 0)
  where
    mod2 :: k Int Int
    mod2 = strong mod' (num 2)
```

```
>>> isEven 2
True

>>> isEven 3
False
```

```haskell
tag :: k (Bool, a) (Either a a)

matchOn :: (Cartesian k, Strong k, Cocartesian k)
        => k a Bool -> k a (Either a a)
matchOn predicate = copy >>> first' predicate >>> tag
```

```
collatzStep :: forall k.
               ( Numeric k
               , Cartesian k
               , Cocartesian k
               , Choice k
               , Strong k
               , MyPrimitives k)
               => k Int Int
collatzStep =
    matchOn isEven
    >>> (onOdds +++ onEvens)
    >>> unify
  where
    onOdds  :: k Int Int
    onOdds = strong mult (num 3) >>> strong add (num 1)
    onEvens :: k Int Int
    onEvens = strong div' (num 2)
```

```
collatzStep :: forall k.
               ( Numeric k
               , Cartesian k
               , Cocartesian k
               , Choice k
               , Strong k
               , MyPrimitives k)
               => k Int Int
collatzStep =
    matchOn isEven
    >>> (onOdds +++ onEvens)
    >>> unify
  where
    onOdds  :: k Int Int
    onOdds = strong mult (num 3) >>> strong add (num 1)
    onEvens :: k Int Int
    onEvens = strong div' (num 2)
```

```
collatzStep :: forall k.
               ( Numeric k
               , Cartesian k
               , Cocartesian k
               , Choice k
               , Strong k
               , MyPrimitives k)
               => k Int Int
collatzStep =
    matchOn isEven
    >>> (onOdds +++ onEvens)
    >>> unify
  where
    onOdds  :: k Int Int
    onOdds = strong mult (num 3) >>> strong add (num 1)
    onEvens :: k Int Int
    onEvens = strong div' (num 2)
```

```
collatzStep :: forall k.
               ( Numeric k
               , Cartesian k
               , Cocartesian k
               , Choice k
               , Strong k
               , MyPrimitives k)
               => k Int Int
collatzStep =
    matchOn isEven
    >>> (onOdds +++ onEvens)
    >>> unify
  where
    onOdds  :: k Int Int
    onOdds = strong mult (num 3) >>> strong add (num 1)
    onEvens :: k Int Int
    onEvens = strong div' (num 2)
```

```
>>> collatzStep 3
10
>>> collatzStep 4
2
>>> collatzStep 5
16
```

```
>>> T.putStrLn $ renderJS collatzStep
```

```
(input) => {
  const fst = (input) => {
  const fst = ((x) => ([x, x]));
  const snd = (input) => {
  const fst = ([l, r]) => {
  const inner = (input) => {
  const fst = (input) => {
  const fst = ((x) => ([x, x]));
  const snd = (input) => {
  const fst = ([l, r]) => {
  const inner = (x) => (0);
  const result = inner(l);
  return [result, r];
};
  const snd = (input) => {
  const fst = (([x, y]) => ([y, x]));
  const snd = (input) => {
  const fst = ([l, r]) => {
  const inner = (input) => {
  const fst = ((x) => ([x, x]));
  const snd = (input) => {
  const fst = (input) => {
  const fst = (([x, y]) => ([y, x]));
  const snd = (input) => {
  const fst = ([l, r]) => {
  const inner = (x) => (2);
  const result = inner(l);
  return [result, r];
};
  const snd = (([x, y]) => ([y, x]));
  return snd(fst(input));
};
  return snd(fst(input));
};
  const snd = ([x, y]) => (x % y);
  return snd(fst(input));
};
  return snd(fst(input));
};
```

```
  const result = inner(l);
  return [result, r];
};
  const snd = (([x, y]) => ([y, x]));
  return snd(fst(input));
};
  return snd(fst(input));
};
  return snd(fst(input));
};
  return snd(fst(input));
};
  const snd = (([x, y]) => x === y);
  return snd(fst(input));
};
  const result = inner(l);
  return [result, r];
};
  const snd = (([b, x]) =>
    ({tag: b ? 'right' : 'left', value: x}));
  return snd(fst(input));
};
  return snd(fst(input));
};
  const snd = (input) => {
  const fst = (input) => {
  const fst = (input) => {
  const overLeft = (input) => {
  const fst = (input) => {
  const fst = ((x) => ([x, x]));
  const snd = (input) => {
  const fst = (input) => {
  const fst = (([x, y]) => ([y, x]));
  const snd = (input) => {
  const fst = ([l, r]) => {
  const inner = (x) => (3);
  const result = inner(l);
  return [result, r];
};
  const snd = (([x, y]) => ([y, x]));
  return snd(fst(input));
};
  return snd(fst(input));
};
```

```
  const snd = ([x, y]) => (x * y);
  return snd(fst(input));
};
  return snd(fst(input));
};
  const snd = (input) => {
  const fst = ((x) => ([x, x]));
  const snd = (input) => {
  const fst = (input) => {
  const fst = (([x, y]) => ([y, x]));
  const snd = (input) => {
  const fst = ([l, r]) => {
  const inner = (x) => (1);
  const result = inner(l);
  return [result, r];
};
  const snd = (([x, y]) => ([y, x]));
  return snd(fst(input));
};
  return snd(fst(input));
};
  const snd = ([x, y]) => (x + y);
  return snd(fst(input));
};
  return snd(fst(input));
};
  return snd(fst(input));
};
  if (input.tag == 'left') {
    return { tag: 'left', value: overLeft(input.value) };
  }
  return input
};
  const snd = (input) => {
  const fst = (({tag, value}) =>
    ({tag: tag === 'left' ? 'right' : 'left', value}));
  const snd = (input) => {
  const fst = (input) => {
  const overLeft = (input) => {
  const fst = ((x) => ([x, x]));
```

```
  const snd = (input) => {
  const fst = (input) => {
  const fst = (([x, y]) => ([y, x]));
  const snd = (input) => {
  const fst = ([l, r]) => {
  const inner = (x) => (2);
  const result = inner(l);
  return [result, r];
};
  const snd = (([x, y]) => ([y, x]));
  return snd(fst(input));
};
  return snd(fst(input));
};
  const snd = ([x, y]) => (x / y);
  return snd(fst(input));
};
  return snd(fst(input));
};
  if (input.tag == 'left') {
    return { tag: 'left',
             value: overLeft(input.value) };
  }
  return input
};
  const snd = (({tag, value}) =>
    ({tag: tag === 'left' ? 'right' : 'left', value}));
  return snd(fst(input));
};
  return snd(fst(input));
};
  return snd(fst(input));
};
  const snd = (x => (x.value));
  return snd(fst(input));
};
  return snd(fst(input));
}
```

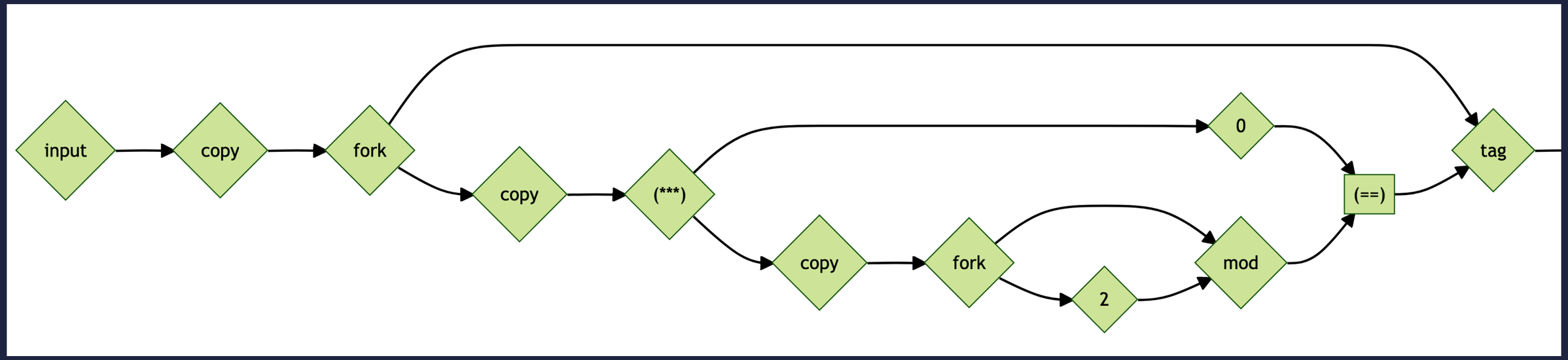
```
> collatzStep(3)
<· 10
> collatzStep(4)
<· 2
> collatzStep(5)
<· 16
>
```

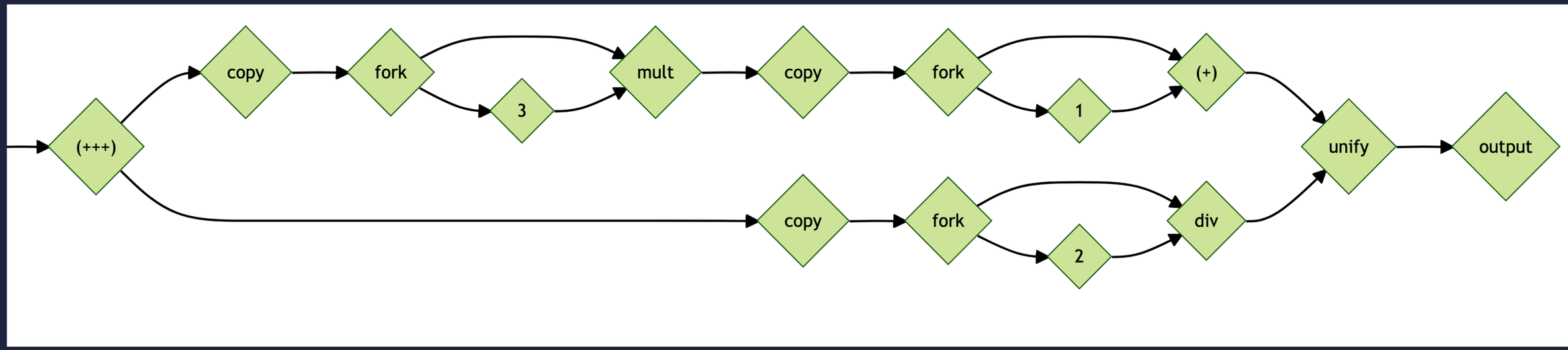
Remember...
This is alpha level software

INTERPRETING AS

Diagrams

```
newtype Diagram a b =
    Diagram {toGraph :: State Graph InputOutputLinks }

>>> renderDiagram (collatzStep  @Diagram)
```


input
copy
fork
copy
(***)
0
copy
fork
2
mod
(==)
tag


(+++)
copy
fork
3
mult
copy
fork
1
(+)
copy
fork
2
div
unify
output

Most diagrams you've seen were made this way!

High level view of any function!

LIGHTNING ROUND

Functions can swap arguments around

```
class Category k => Symmetric k where
  swap       :: k (l, r) (r, l)
  swapE      :: k (Either l r) (Either r l)

  reassoc    :: k (a, (b, c)) ((a, b), c)
  reassocE   :: k (Either a (Either b c)) (Either (Either a b) c)
```

```
class Category k => Symmetric k where
  swap       :: k (l, r) (r, l)
  swapE      :: k (Either l r) (Either r l)

  reassoc    :: k (a, (b, c)) ((a, b), c)
  reassocE   :: k (Either a (Either b c)) (Either (Either a b) c)
```

```
class Category k => Symmetric k where
  swap       :: k (l, r) (r, l)
  swapE      :: k (Either l r) (Either r l)

  reassoc    :: k (a, (b, c)) ((a, b), c)
  reassocE   :: k (Either a (Either b c)) (Either (Either a b) c)
```

```haskell
class Category k => Symmetric k where
  swap       :: k (l, r) (r, l)
  swapE      :: k (Either l r) (Either r l)

  reassoc    :: k (a, (b, c)) ((a, b), c)
  reassocE   :: k (Either a (Either b c)) (Either (Either a b) c)
```

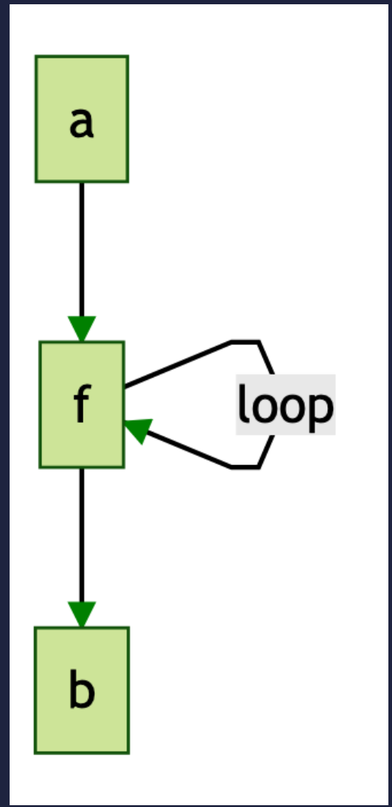
FUNCTIONS CAN RECURSE

COCHOICE

```
Cochoice k where
  unleft :: k (Either a c) (Either b c)
         -> k a b
```

COCHOICE

```
unleft (f :: Either a c -> Either b c)
```



COSTRONG

```
Costrong k where
  unfirst :: k (a, c) (b, c)
          -> k a b
```

WE CAN MODEL

composition
branching
copying
routing
recursion
fixpoints

Can also encode classes for

`Reader`

`Writer`

`State`

`Cont`

`IO`

ACT IV.

FUNCTIONS AS DATA

FREE FUNCTIONS

FREE FUNCTIONS

```
data FreeFunc p a b where
  Id      :: FreeFunc p x x
  Compose :: FreeFunc p x y -> FreeFunc p y z -> FreeFunc p x z
  Copy    :: FreeFunc p x (x, x)
  Consume :: FreeFunc p x ()
  First   :: FreeFunc p a b -> FreeFunc p (a, x) (b, x)
  Second  :: FreeFunc p a b -> FreeFunc p (x, a) (x, b)
  Fst     :: FreeFunc p (a, x) a
  Snd     :: FreeFunc p (x, a) a
  Lift    :: p a b -> FreeFunc p a b
```

This version implements Category, Cartesian, and Strong;
but it works for all the other classes too.

```haskell
instance Category (FreeFunc k) where
  id = Id
  (.) = flip Compose

instance Cartesian (FreeFunc k) where
  copy = Copy
  consume = Consume
  fst' = Fst
  snd' = Snd

instance Strong (FreeFunc k) where
  first'  = First
  second' = Second
```

GENERALIZED OVER A LAYER OF PRIMITIVES

👇

```
data FreeFunc p a b where
  Id      :: FreeFunc p x x
  Compose :: FreeFunc p x y -> FreeFunc p y z -> FreeFunc p x z
  Copy    :: FreeFunc p x (x, x)
  Consume :: FreeFunc p x ()
  First   :: FreeFunc p a b -> FreeFunc p (a, x) (b, x)
  Second  :: FreeFunc p a b -> FreeFunc p (x, a) (x, b)
  Fst     :: FreeFunc p (a, x) a
  Snd     :: FreeFunc p (x, a) a
  Lift    :: p a b -> FreeFunc p a b
```

```haskell
data Prims a b where
  ReverseString :: Prims String String
  Equal :: Eq a => Prims (a, a) Bool

instance MyPrimitives (FreeFunc Prims) where
  reverseString = Lift ReverseString
  eq            = Lift Equal
```

```
isPalindrome :: (MyPrimitives k, Cartesian k, Strong k)
             => k String Bool
isPalindrome =
        copy
    >>> first' reverse'
    >>> eq
```

We can encode this function as data!

```haskell
{-# LANGUAGE QuantifiedConstraints #-}
{-# LANGUAGE StandaloneDeriving #-}

deriving instance Show (Prims a b)
deriving instance (forall a b. Show (p a b)) => Show (FreeFunc p x y)

>>> show (isPalindrome @(FreeFunc Prims ))
"Compose Copy (Compose (First (Lift ReverseString)) (Lift Equal))"
```

```haskell
{-# LANGUAGE QuantifiedConstraints #-}
{-# LANGUAGE StandaloneDeriving #-}

deriving instance Show (Prims a b)
deriving instance (forall a b. Show (p a b)) => Show (FreeFunc p x y)

>>> show (isPalindrome @(FreeFunc Prims ))
"Compose Copy (Compose (First (Lift ReverseString)) (Lift Equal))"
```

```haskell
{-# LANGUAGE QuantifiedConstraints #-}
{-# LANGUAGE StandaloneDeriving #-}

deriving instance Show (Prims a b)
deriving instance (forall a b. Show (p a b)) => Show (FreeFunc p x y)

>>> show (isPalindrome @(FreeFunc Prims ))
"Compose Copy (Compose (First (Lift ReverseString)) (Lift Equal))"
```

Since our function is just data
We can serialize it too!

```haskell
instance ToJSON (Prims a b) where
  toJSON
    = \case
        ReverseString -> String "ReverseString"
        Equal -> String "Equal"

instance (forall a b. ToJSON (k a b)) => ToJSON (FreeFunc k x y) where
  toJSON
    = \case
        Id -> String "Id"
        (Compose l r) -> Array [String "Compose", toJSON l, toJSON r]
        Copy -> String "Copy"
        Consume -> String "Compose"
        (First l) -> Array [String "First", toJSON l]
        (Second l) -> Array [String "Second", toJSON l]
        Fst -> String "Fst"
        Snd -> String "Snd"
        (Lift p) -> Array [String "Lift", toJSON p]
```

```
instance ToJSON (Prims a b) where
  toJSON
    = \case
        ReverseString -> String "ReverseString"
        Equal -> String "Equal"

instance (forall a b. ToJSON (k a b)) => ToJSON (FreeFunc k x y) where
  toJSON
    = \case
        Id -> String "Id"
        (Compose l r) -> Array [String "Compose", toJSON l, toJSON r]
        Copy -> String "Copy"
        Consume -> String "Compose"
        (First l) -> Array [String "First", toJSON l]
        (Second l) -> Array [String "Second", toJSON l]
        Fst -> String "Fst"
        Snd -> String "Snd"
        (Lift p) -> Array [String "Lift", toJSON p]
```

```
>>> BL.putStrLn $ encode (isPalindrome @(FreeFunc Prims))
[
  "Compose",
      "Copy",
      [ "Compose",
          [ "First",
              [ "Lift", "ReverseString" ]
          ],
          [ "Lift", "Equal" ]
      ]
]
```

Deserializing functions in Haskell
requires re-interpreting type information
from a JSON value

Would be much easier with dependant types

But other languages aren't so strict!

DESERIALIZING FUNCTIONS

We can safely send/accept functions over the wire

DESERIALIZING FUNCTIONS

We can check for possible divergence by inspecting structure for *strong/costrong*

DESERIALIZING FUNCTIONS

You can interpret the function iteratively
to limit run-time or implement time-sharing

APACHE BEAM / HADOOP

OPTIMIZING CATEGORIES

IN Summary

Functions are a conglomeration of many behaviours

Using too much power limits our options

We can do more with code than execute it

There are interesting categories that aren't programs

SOME IDEAS

Verilog
Minecraft Redstone
GPUs
SMT/SAT solvers
Probabilistic analysis

Non-determinism
Linear Arrows
FRP
State Machines
Free Parallelization

We can encode functions as data...

...which can be analyzed, optimized, or serialized

The syntax could still use some work

But the principles are sound

Conal Elliot's

COMPILING TO CATEGORIES

HOPE YOU LEARNED SOMETHING 👋

CHRIS PENNER

chrispenner.ca

github.com/ChrisPenner

@ChrisLPenner

Coupon code:
leanpub.com/optics-by-example/c/berlin