# Unstructured Recursion

# Recursion

```
fac :: Int -> Int
fac n = if n == 0
        then 1
        else fac (n - 1)
```

# Recursion

```
fac :: Int -> Int
fac n = if n == 0
        then 1
        else fac (n - 1)
```

# Explicit Recursion

```
fac = fix fac'
```

# Explicit Recursion

non-recursive definition

```
fac = fix fac'
```

# Explicit Recursion

non-recursive definition

```
fac = fix fac'
```

generic fixpoint combinator

# Non-Recursive Version

```
fac' f n = if n == 0
           then 1
           else f (n - 1)
```

1 level of the recursion

# Non-Recursive Version

```
fac' f n = if n == 0
           then 1
           else f (n - 1)
```

1 level of the recursion

parameter for the other levels

# Non-Recursive Version

```
fac' :: (Int -> Int) -> (Int -> Int)
fac' f n = if n == 0
           then 1
           else f (n - 1)
```

1 level of the recursion

parameter for the other levels

# Fixpoint Combinator

```
fix :: (a -> a) -> a
```

# Fixpoint Combinator

expression with hole

```
fix :: (a -> a) -> a
```

# Fixpoint Combinator

expression with hole

hole plugged with itself

```
fix :: (a -> a) -> a
```

# Fixpoint Combinator

expression with hole

hole plugged with itself

```
fix :: (a -> a) -> a
fix f = f (fix f)
```

# Fixpoint Combinator

expression with hole

hole plugged with itself

```
fix :: (a -> a) -> a
fix f = f (fix f)
      = f (f (fix f))
```

# Fixpoint Combinator

expression with hole

hole plugged with itself

```
fix :: (a -> a) -> a
fix f = f (fix f)
      = f (f (fix f))
      = f (f (f (fix f)))
```

# Fixpoint Combinator

expression with hole

hole plugged with itself

```
fix :: (a -> a) -> a
fix f = f (fix f)
      = f (f (fix f))
      = f (f (f (fix f)))
      = f (f (f (f …)))
```

f all the way down

# General Recursion

```
fix :: (a -> a) -> a
fix f = f (fix f)
```

# General Recursion

```
fix :: (a -> a) -> a
fix f = f (fix f)
```

makes language
Turing complete

# General Recursion

```
fix :: (a -> a) -> a
fix f = f (fix f)
```

can express all recursion

makes language Turing complete

# General Recursion Considered Harmful

# General Recursion Considered Harmful

```
loop = fix id
```

# General Recursion Considered Harmful

```
loop = fix id
```

```
diverge = fix go
 where
   go f n = f (n+1)
```

# General Recursion Considered Harmful

```
loop = fix id
```

Easy to go over the egde

```
diverge = fix go
 where
   go f n = f (n+1)
```

# Recursion Schemes

# A handrail to hold on to

```
data List
  = Nil
  | Cons Int List
```

# A handrail to hold on to

```
data List
  = Nil
  | Cons Int List
```

No irregular syntax

# A handrail to hold on to

```
data List
  = Nil
  | Cons Int List
```

No irregular syntax

No type parameter

# A handrail to hold on to

Keeping things simple

```
data List
  = Nil
  | Cons Int List
```

No irregular syntax

No type parameter

# Recursion follows Recursion

```
sum Nil              = 0
sum (Cons x xs) = x + sum xs
```

```
data List  = Nil
           | Cons Int List
```

# Sliding down the railing

```
sum Nil              = 0
sum (Cons x xs) = x + sum xs

prod Nil             = 1
prod (Cons x xs) = x * prod xs
```

# Sliding down the railing

```
sum Nil              = 0
sum (Cons x xs) = x + sum xs

prod Nil             = 1
prod (Cons x xs) = x * prod xs
```

# Sliding down the railing

```
sum Nil              = 0
sum (Cons x xs) = x + sum xs

prod Nil             = 1
prod (Cons x xs) = x * prod xs
```

# Fold Recursion Scheme

```
sum Nil              = 0
sum (Cons x xs) = x + sum xs

prod Nil             = 1
prod (Cons x xs) = x * prod xs
```

**Abstract**

```
fold n c Nil              = n
fold n c (Cons x xs) = c x (fold n c xs)
```

# Fold Recursion Scheme

```
sum Nil              = 0
sum (Cons x xs) = x + sum xs

prod Nil             = 1
prod (Cons x xs) = x * prod xs
```

Abstract

```
fold n c Nil              = n
fold n c (Cons x xs) = c x (fold n c xs)
```

# Fold Recursion Scheme

```
sum Nil              = 0
sum (Cons x xs)      = x + sum xs

prod Nil             = 1
prod (Cons x xs)     = x * prod xs
```

Abstract

```
fold n c Nil              = n
fold n c (Cons x xs)      = c x (fold n c xs)
```

# Fold Recursion Scheme

```
sum = fold 0 (+)

prod = fold 1 (*)

fold n c Nil              = n
fold n c (Cons x xs) = c x (fold n c xs)
```

# Structured Recursion

```
fold n c Nil
  = n
fold n c (Cons x xs)
  = c x (fold n c xs)
```

# Structured Recursion

```
fold n c Nil
  = n
fold n c (Cons x xs)
  = c x (fold n c xs)
```

Recursion follows list structure

# Structured Recursion

```
fold n c Nil
  = n
fold n c (Cons x xs)
  = c x (fold n c xs)
```

Recursion follows list structure

Recursive call over recursive list

# Structured Recursion

```
fold n c Nil
  = n
fold n c (Cons x xs)
  = c x (fold n c xs)
```

Recursion follows list structure

Recursive call over recursive list

Terminates (for finite list)

# Recursive Types

# Explicit Recursion

```
fac = fix fac'
```

# Explicit Recursion

non-recursive definition

```
fac = fix fac'
```

# Explicit Recursion

non-recursive definition

```
fac = fix fac'
```

generic fixpoint combinator

# Explicit Recursion

```
fac = fix fac'
```

# Explicit Recursion

```
fac = fix fac'
```

```
List <~> Fix List'
```

# Explicit Recursion

```
fac = fix fac'
```

```
List <~> Fix List'
```

generic fixpoint combinator

# Explicit Recursion

```
fac = fix fac'
```

```
List <~> Fix List'
```

non-recursive definition

generic fixpoint combinator

# Fixpoint Combinator

```
fix :: (a -> a) -> a
fix f = f (fix f)
```

# Fixpoint Combinator

```
fix :: (a -> a) -> a
fix f = f (fix f)
```

# Fixpoint Combinator

```
fix :: (a -> a) -> a
fix f = f (fix f)
```

```
-- Fix :: (* -> *) -> *
type Fix f = f (Fix f)
```

# Fixpoint Combinator

```
fix :: (a -> a) -> a
fix f = f (fix f)
```

Recursive Type Synonyms not supported

```
-- Fix :: (* -> *) -> *
type Fix f = f (Fix f)
```

# Fixpoint Combinator

```
fix :: (a -> a) -> a
fix f = f (fix f)
```

```
-- Fix :: (* -> *) -> *
data Fix f = In (f (Fix f))
```

# Non-Recursive Version

```
-- List :: *
data List  = Nil
           | Cons Int List
```

# Non-Recursive Version

```
-- List :: *
data List   = Nil
            | Cons Int List
```

```
-- List' :: * -> *
data List' r   = Nil'
               | Cons' Int r
```

# Isomorphism

```
iso :: List <~> Fix List'
iso = Iso t f where
  t Nil              = In Nil'
  t (Cons x xs) = In (Cons' x (t xs))

  f (In Nil')                = Nil
  f (In (Cons' x xs)) = Cons x (f xs)
```

# Generic Fold

# What fold does

```
Cons 1 (Cons 2 (Cons 3 Nil)))
```

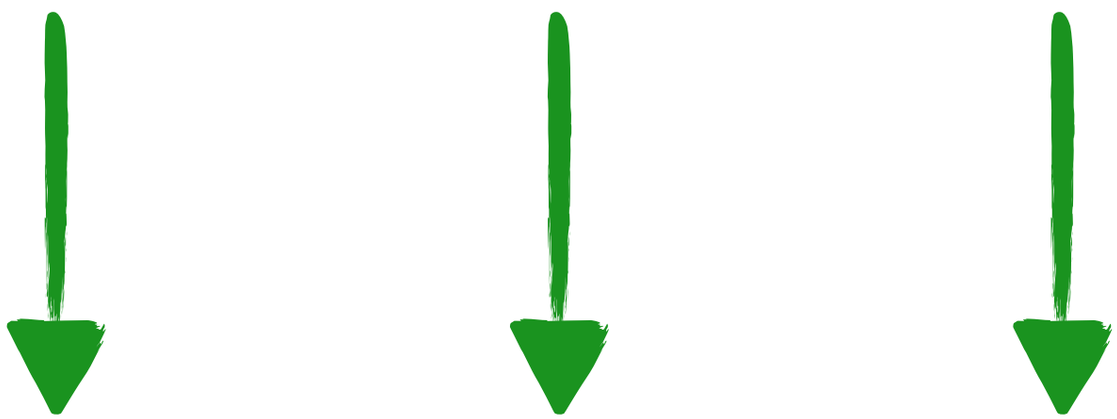
```
fold 0 (+)
```

# What fold does

```
Cons 1 (Cons 2 (Cons 3 Nil)))

(+)  1 ((+)  2 ((+)  3 0  )))

fold 0 (+)
```

# What fold does

```
Cons 1 (Cons 2 (Cons 3 Nil)))
```

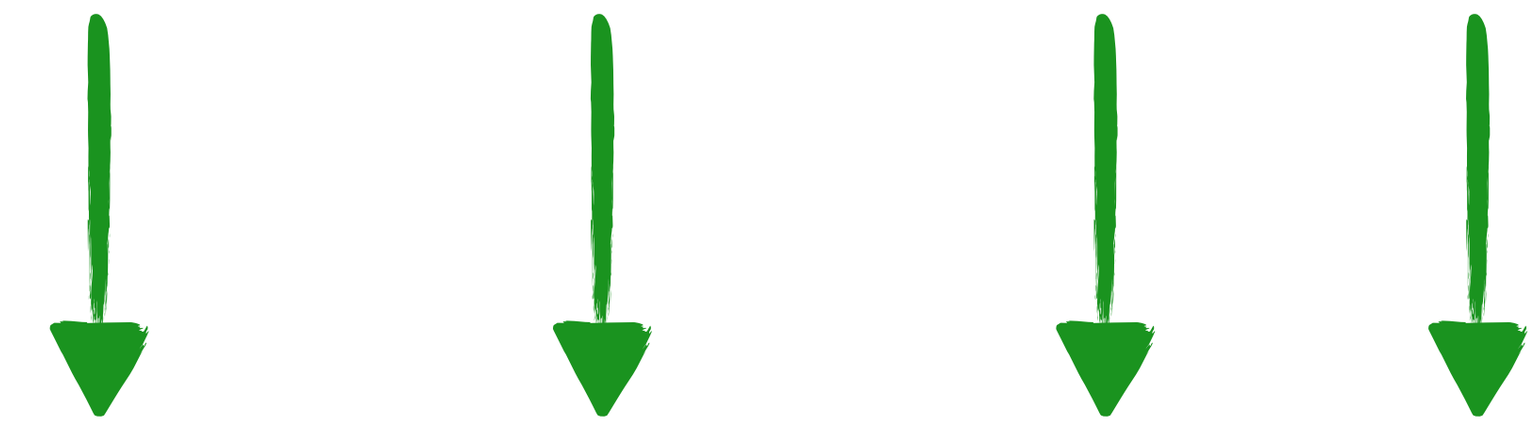


```
(+)   1 ((+)   2 ((+)   3 0   )))
```

```
fold 0 (+)
```

# What fold does

```
Cons 1 (Cons 2 (Cons 3 Nil)))
```



```
(+)   1 ((+)   2 ((+)   3 0   )))
```

```
fold 0 (+)
```

# The Type of Fold

```
fold
  :: a -> (Int -> a -> a) -> List -> a
fold n c Nil
  = n
fold n c (Cons x xs)
  = c x (fold n c xs)
```

# The Type of Fold

```
fold
  :: a -> (Int -> a -> a) -> List -> a
fold n c Nil
  = n
fold n c (Cons x xs)
  = c x (fold n c xs)
```

# The List Algebra

# The List Algebra

```
a -> (Int -> a -> a) -> …
```

# The List Algebra

```
a -> (Int -> a -> a) -> …
```

<~>

# The List Algebra

```
a -> (Int -> a -> a) -> …
```

<~>

```
a * (Int -> a -> a) -> …
```

# The List Algebra

```
a -> (Int -> a -> a) -> …
```

<~>

```
a * (Int -> a -> a) -> …
```

<~>

# The List Algebra

```
a -> (Int -> a -> a) -> …
```

<~>

```
a * (Int -> a -> a) -> …
```

<~>

```
(One -> a) * (Int -> a -> a) -> …
```

# The List Algebra

```
a -> (Int -> a -> a) -> …
```

<~>

```
a * (Int -> a -> a) -> …
```

<~>

```
(One -> a) * (Int -> a -> a) -> …
```

<~>

# The List Algebra

```
a -> (Int -> a -> a) -> …
```

<~>

```
a * (Int -> a -> a) -> …
```

<~>

```
(One -> a) * (Int -> a -> a) -> …
```

<~>

```
(One -> a) * (Int * a -> a) -> …
```

# The List Algebra

```
a -> (Int -> a -> a) -> …
                <~>
 a * (Int -> a -> a) -> …
                <~>
 (One -> a) * (Int -> a -> a) -> …
                <~>
 (One -> a) * (Int * a -> a) -> …
                <~>
```

# The List Algebra

```
a -> (Int -> a -> a) -> …
```

<~>

```
a * (Int -> a -> a) -> …
```

<~>

```
(One -> a) * (Int -> a -> a) -> …
```

<~>

```
(One -> a) * (Int * a -> a) -> …
```

<~>

```
((One + Int * a) -> a) -> …
```

# The List Algebra

```
a -> (Int -> a -> a) -> …
                <~>
 a * (Int -> a -> a) -> …
                <~>
 (One -> a) * (Int -> a -> a) -> …
                <~>
 (One -> a) * (Int * a -> a) -> …
                <~>
   ((One + Int * a) -> a) -> …
                <~>
```

# The List Algebra

```
a -> (Int -> a -> a) -> …
```

<~>

```
a * (Int -> a -> a) -> …
```

<~>

```
(One -> a) * (Int -> a -> a) -> …
```

<~>

```
(One -> a) * (Int * a -> a) -> …
```

<~>

```
((One + Int * a) -> a) -> …
```

<~>

```
(List' a -> a) -> …
```

# Towards Generic Fold

```
fold
  :: a -> (Int -> a -> a) -> List -> a
fold n c Nil
  = n
fold n c (Cons x xs)
  = c x (fold n c xs)
```

# Towards Generic Fold

```
fold
  :: (List' a -> a) -> List -> a
fold alg Nil
  = alg Nil'
fold alg (Cons x xs)
  = alg (Cons' x (fold alg xs))
```

# Towards Generic Fold

```
fold
  :: (List' a -> a) -> Fix List' -> a
fold alg (In Nil')
  = alg Nil'
fold alg (In (Cons' x xs))
  = alg (Cons' x (fold alg xs))
```

# Functor

```
class Functor f where
 fmap :: (r -> s) -> (f r -> f s)

instance Functor List' where
 fmap f Nil'            =  Nil'
 fmap f (Cons' x r)     =  Cons' x (f r)
```

# Towards Generic Fold

```
fold
  :: (List' a -> a) -> Fix List' -> a
fold alg (In s)
  = alg (fmap (fold alg) s)
```

# Generic Fold

```
fold
  :: Functor f
  => (f a -> a) -> Fix f -> a
fold alg (In s)
  = alg (fmap (fold alg) s)
```

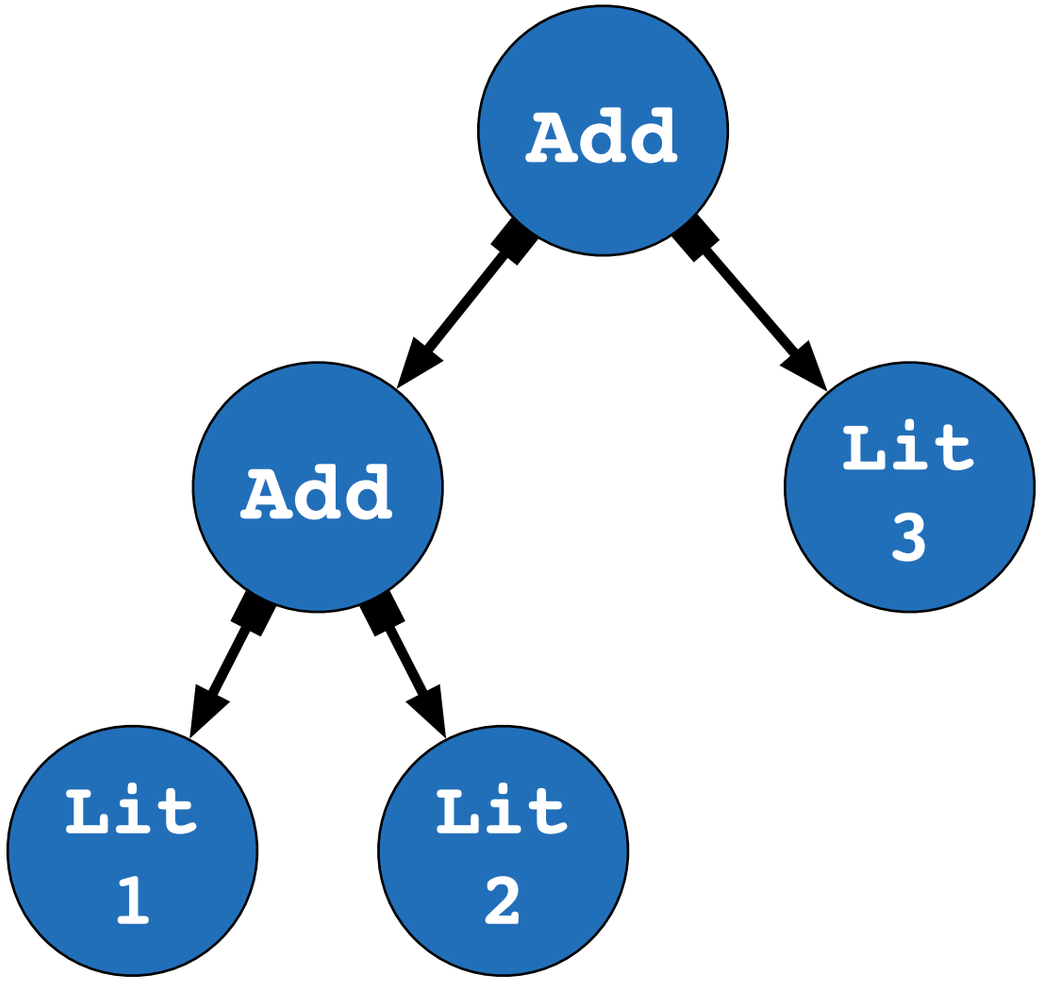
# Example

```
data Exp = Lit Int | Add Exp Exp

data Exp' r = Lit' Int | Add' r r

instance Functor Exp' where
 fmap f (Lit' n)
   = Lit' n
 fmap f (Add' e1 e2)
   = Add' (f e1) (f e2)
```

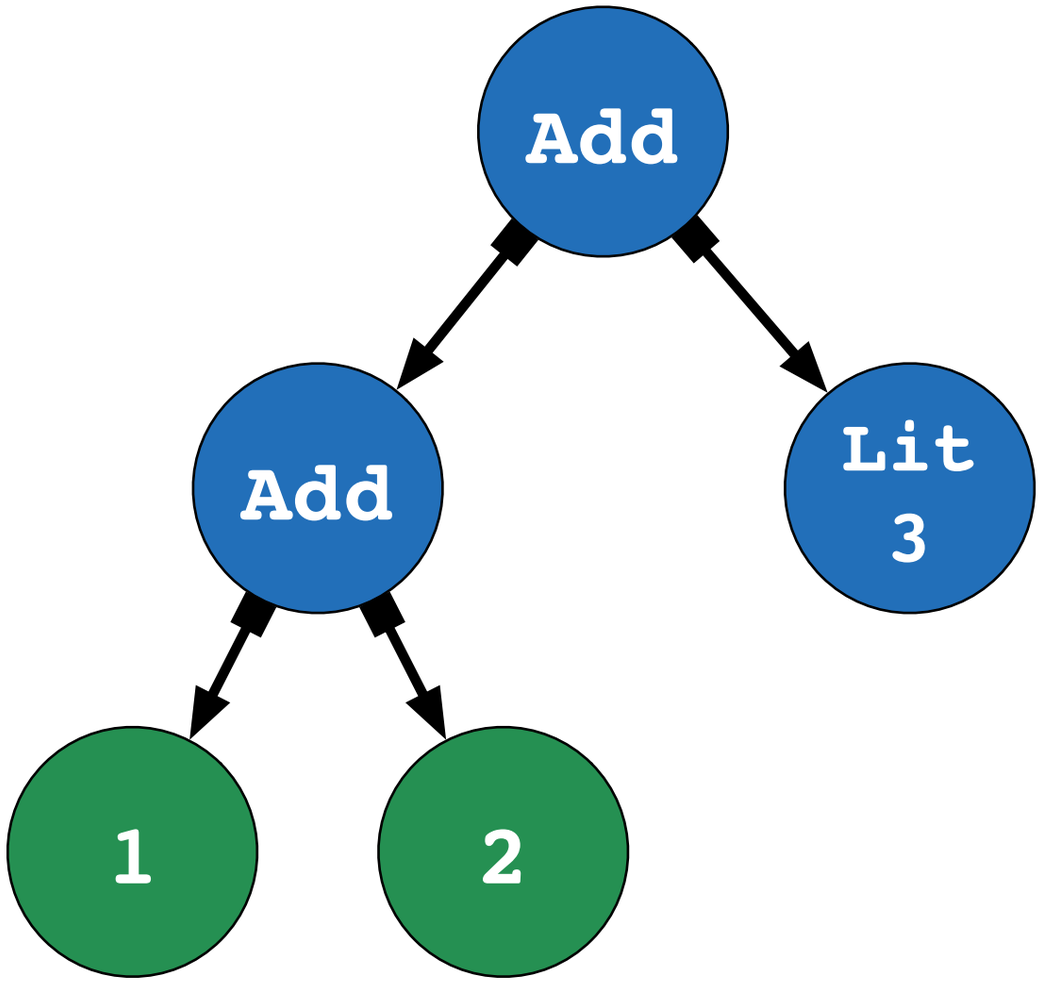
# Example

```
eval :: Fix Exp' -> Int
eval = fold alg where

    alg :: Exp' Int -> Int
    alg (Lit' n)     =  n
    alg (Add' x y)   =  x + y
```

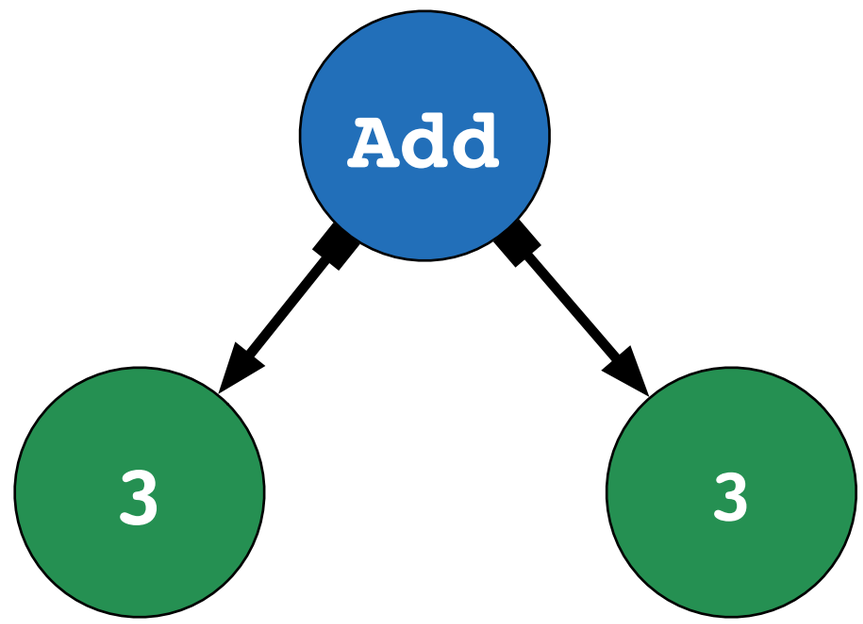
# Example



# Example



# Example



# Example

# Summary

# Fold

- ★ Unstructured Recursion
  - ✦ powerful
  - ✦ dangerous
- ★ Structured Recursion with Fold
  - ✦ safe
  - ✦ often expressive enough
  - ✦ datatype generic concept

# More to Learn

- ★ Fold fusion
  - ✦ parallel fold fusion
  - ✦ fold / build fusion
  - ✦ deforestation
- ★ Other recursion schemes

# Parallel Fold

```
average :: List Float -> Float
average l =
  sum l
    /
  length l
```

# Parallel Fold

```
average :: List Float -> Float
average l =
  fold 0 (+) l
    /
  fold 0 (\_ y -> 1 + y) l
```

# Parallel Fold Fusion

```
average :: List Float -> Float
average l =
  uncurry (/) (fold nil cons l)
  where
    nil              = (0,0)
    cons x (s,l) = (x + s, 1 + l)
```

Jasper Van der Jeugt

# Enabling Fusion

2012
-
2013

GHC compiler

pipeline of recursive functions

Jasper Van der Jeugt

# Enabling Fusion

2012
-
2013

GHC compiler

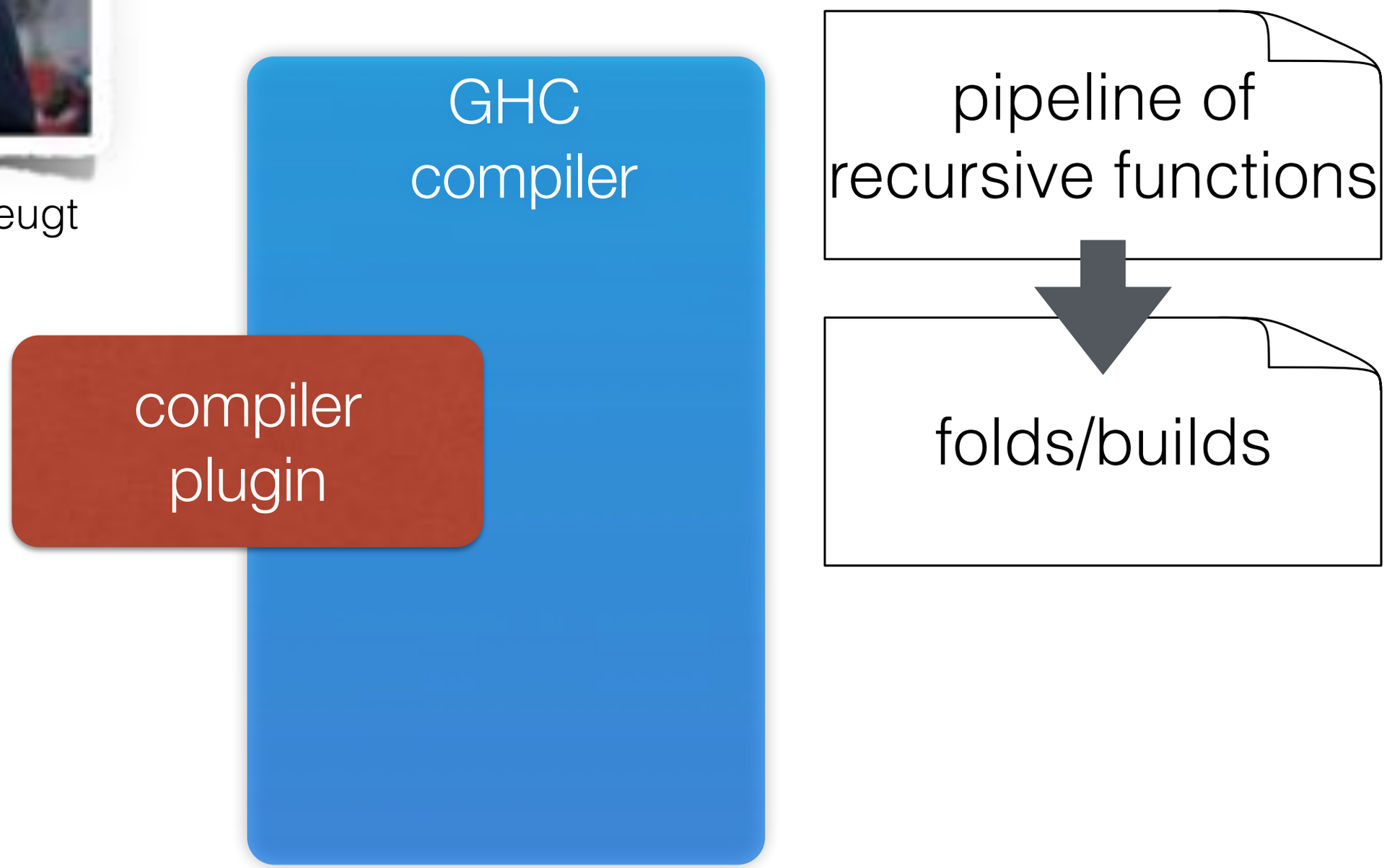
compiler plugin

pipeline of recursive functions

Jasper Van der Jeugt

# Enabling Fusion

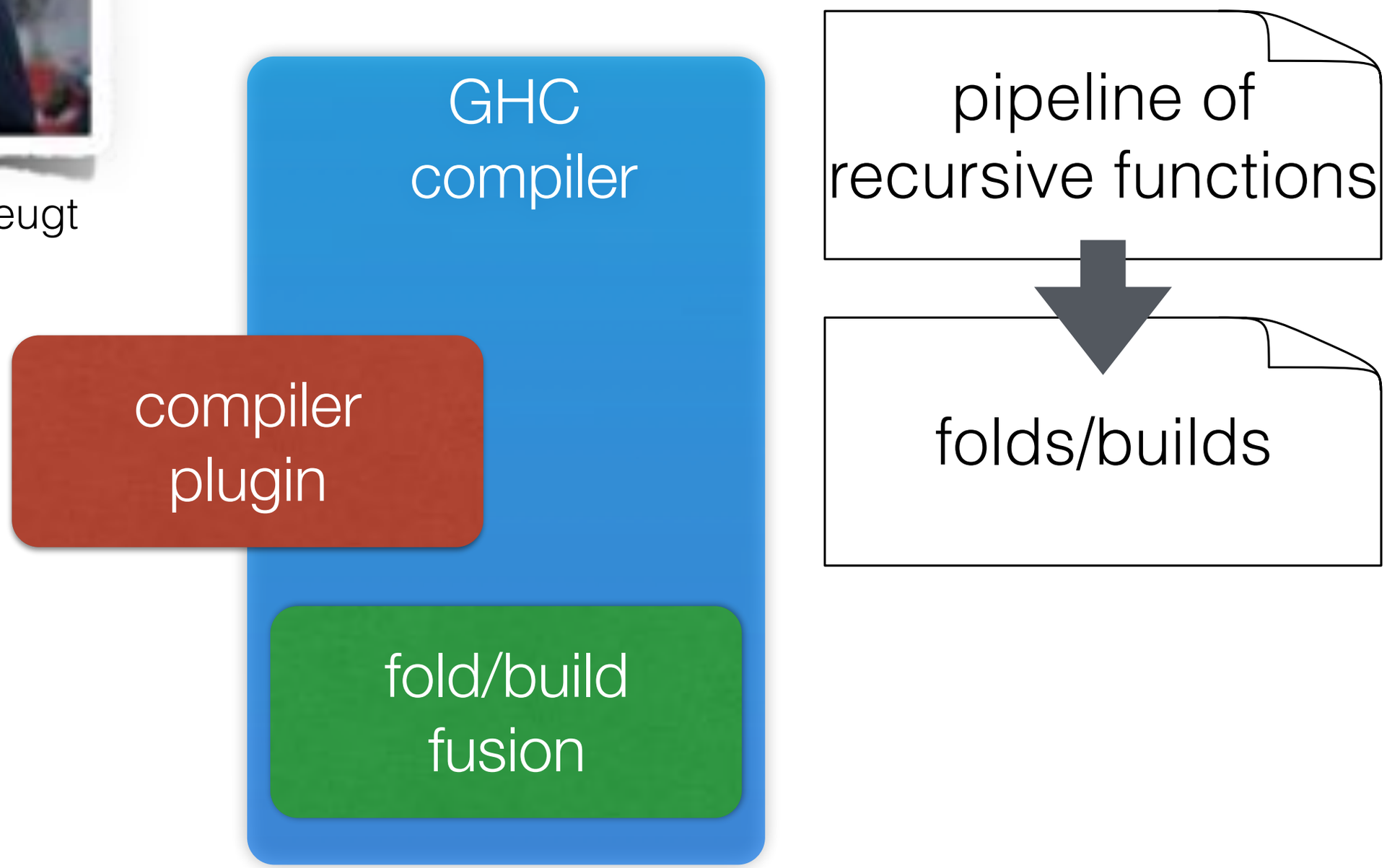
2012
-
2013



Jasper Van der Jeugt

# Enabling Fusion

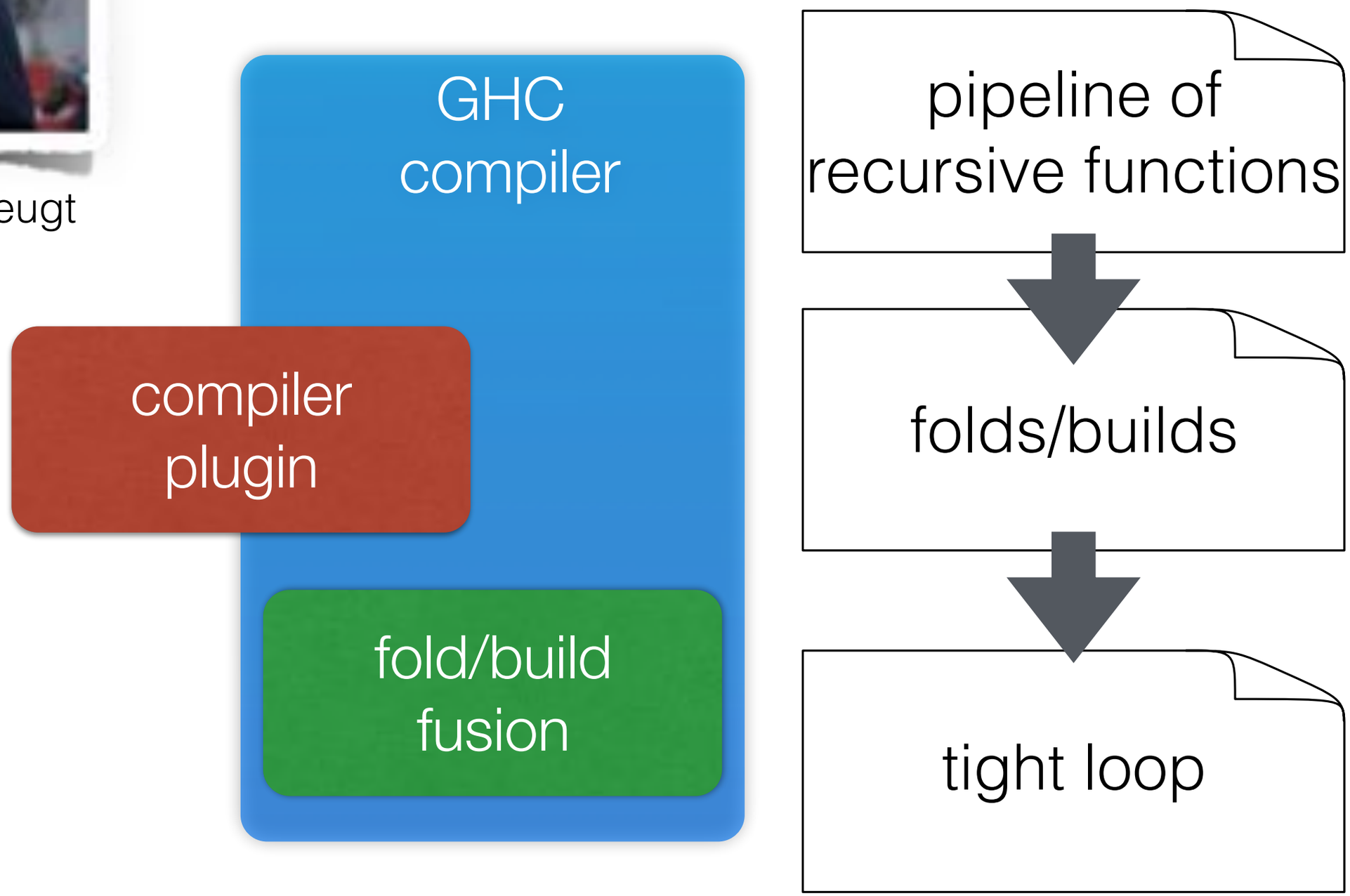
2012
-
2013



Jasper Van der Jeugt

# Enabling Fusion

2012
-
2013



# Enabling Fusion

2012
-
2013

Jasper Van der Jeugt

better.io, Zurich

GHC compiler

compiler plugin

fold/build fusion

pipeline of recursive functions

folds/builds

tight loop

# Fast Monads for Free

2015
-
2016

You

GHC compiler

stack of monads

# Fast Monads for Free

2015
-
2016

You

GHC compiler

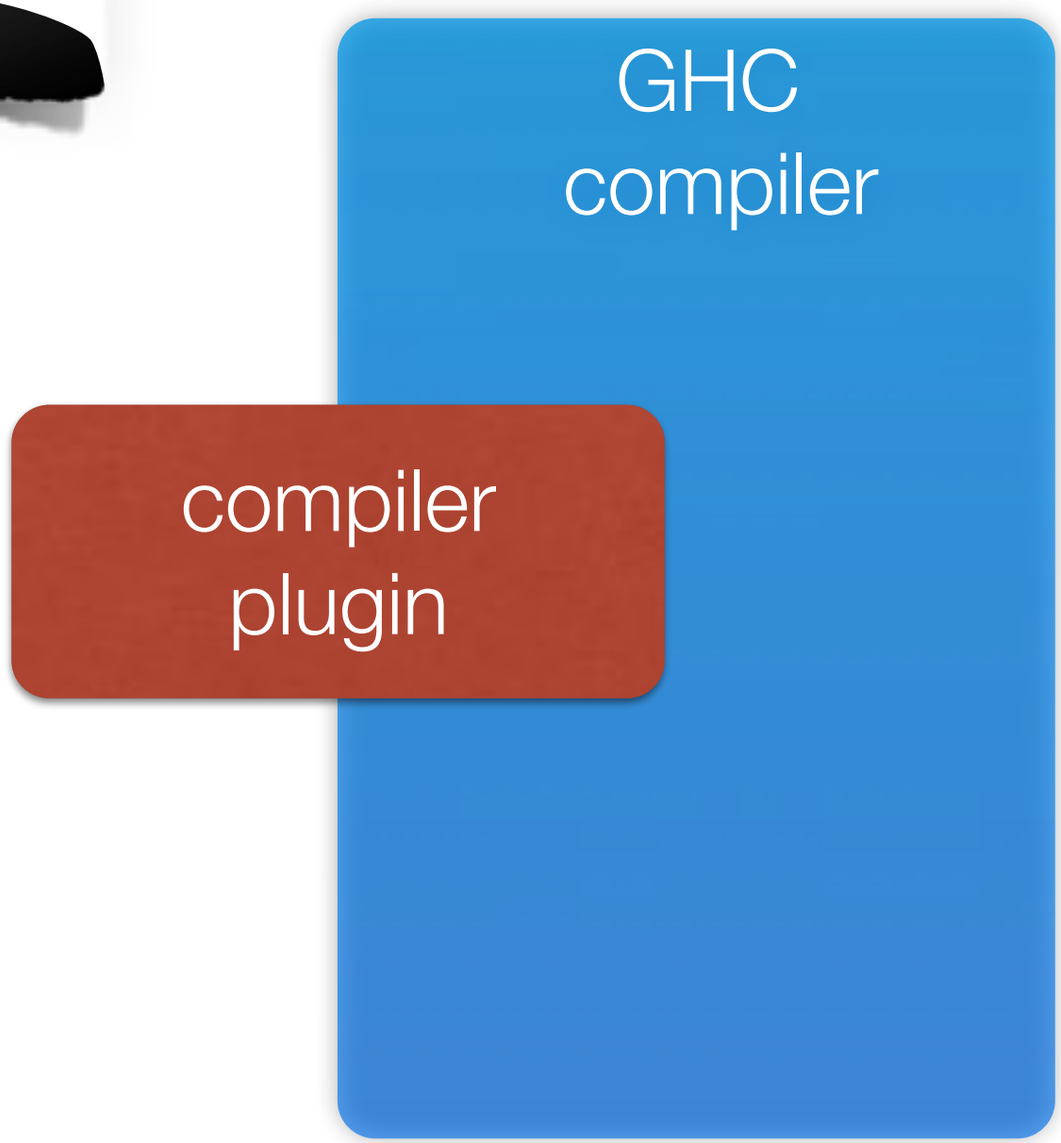
compiler plugin

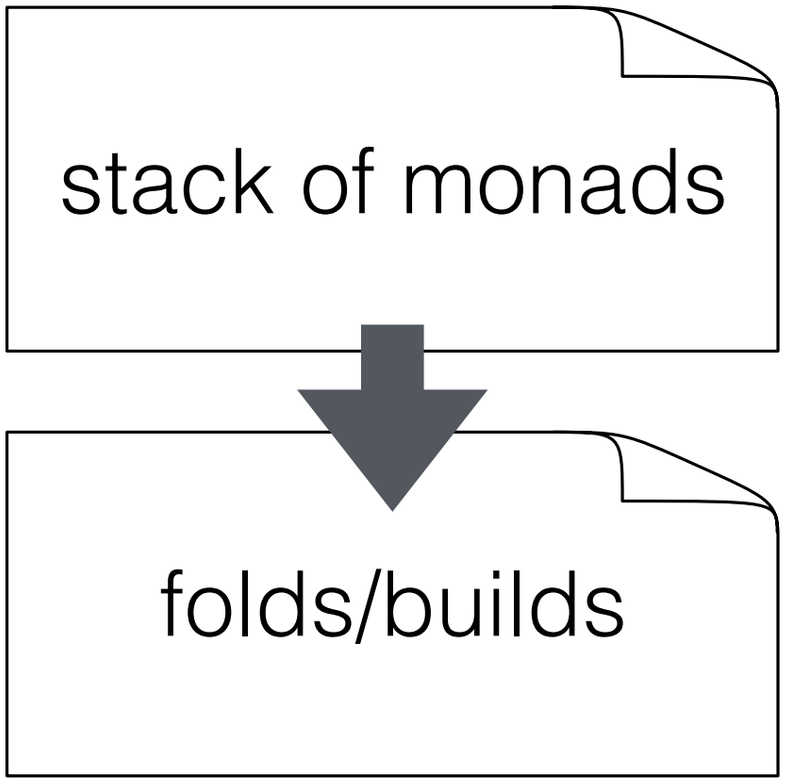
stack of monads

# Fast Monads for Free

2015
-
2016

You

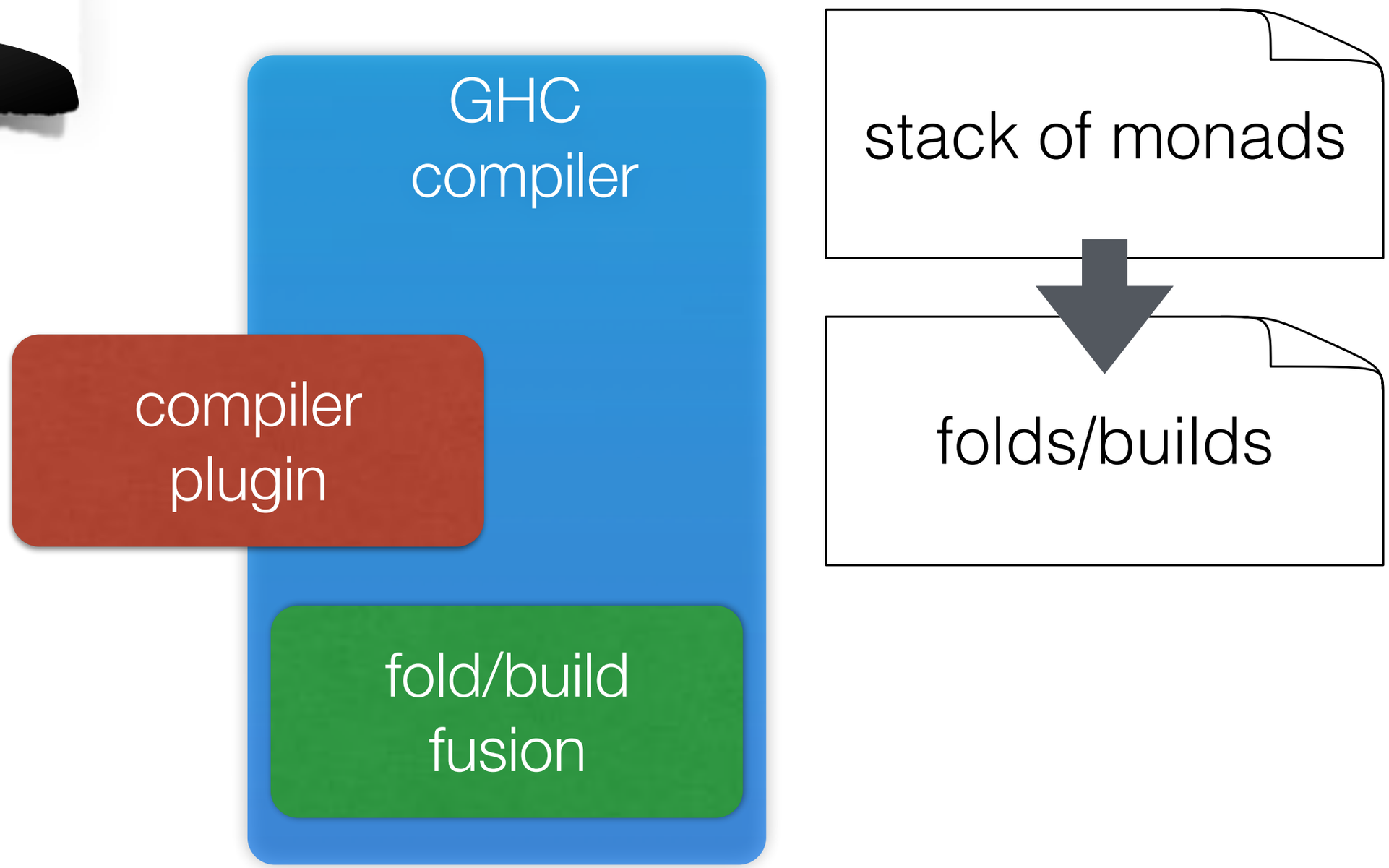


# Fast Monads for Free

2015
-
2016

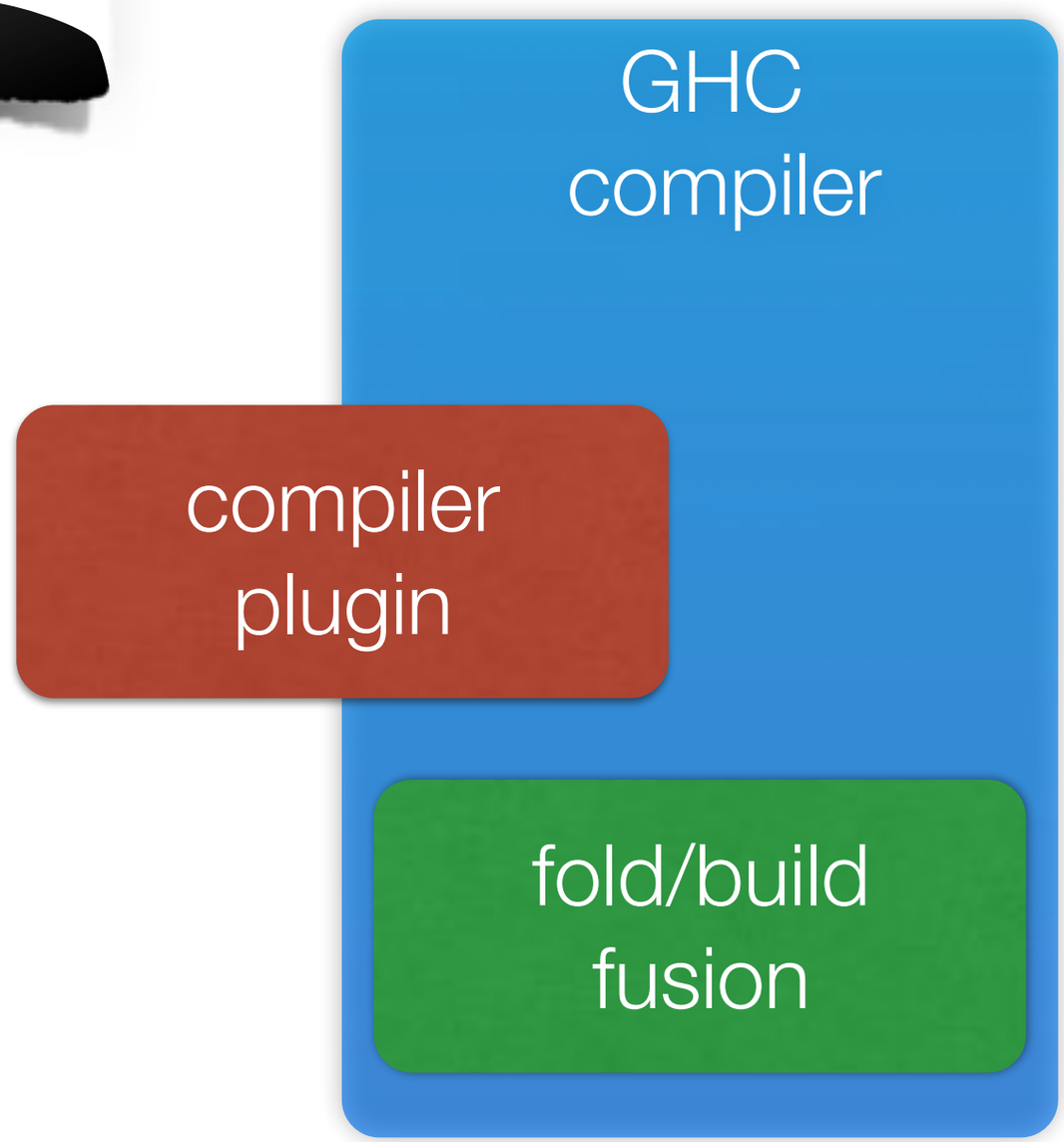
You

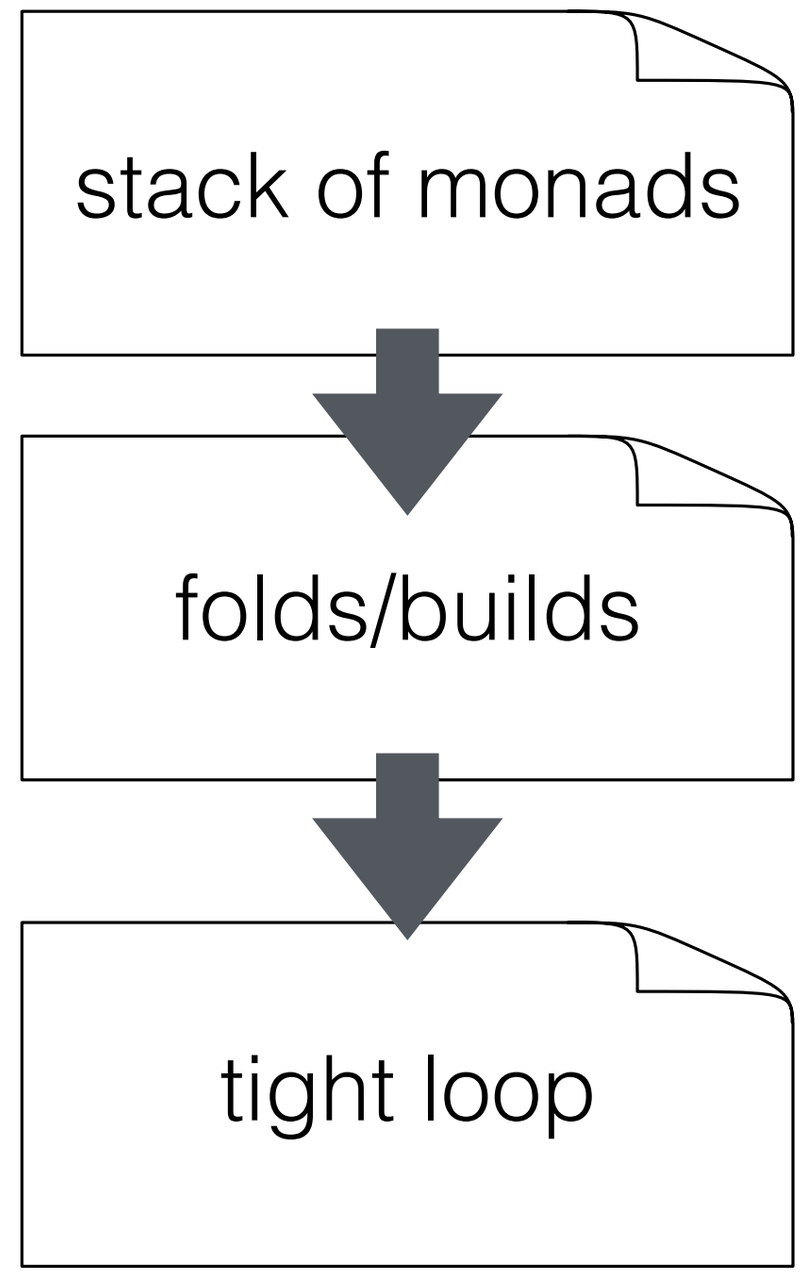


# Fast Monads for Free

2015
-
2016

You

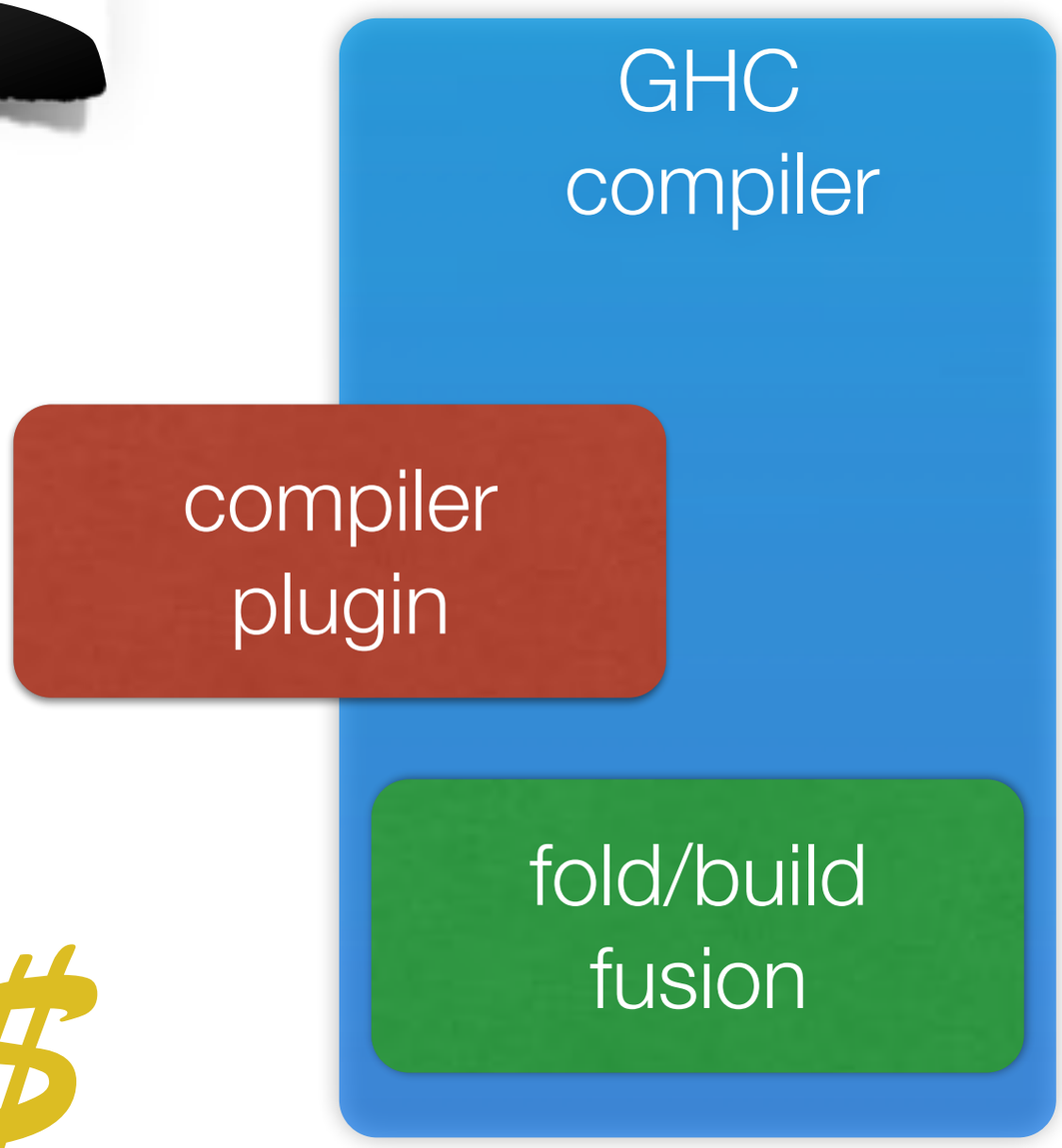


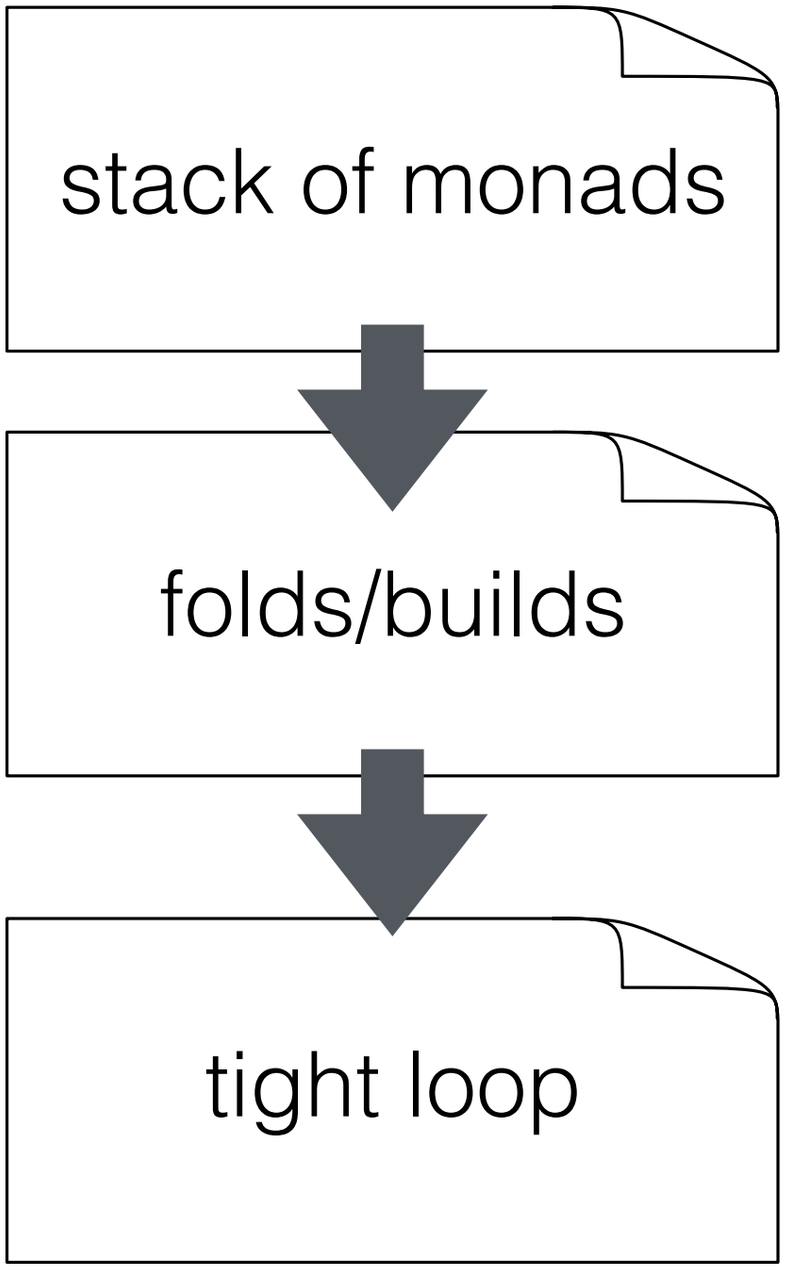



# Fast Monads for Free

2015
-
2016

You




# Cata-morphism


# Cata-morphism

Greek:
*down*


# Cata-morphism

Greek:
*down*

Category theory:
*structure-preserving function*

slide from Nicolas Wu


# ZOO OF MORPHISMS

slide from Nicolas Wu



# ZOO OF MORPHISMS

slide from Nicolas Wu



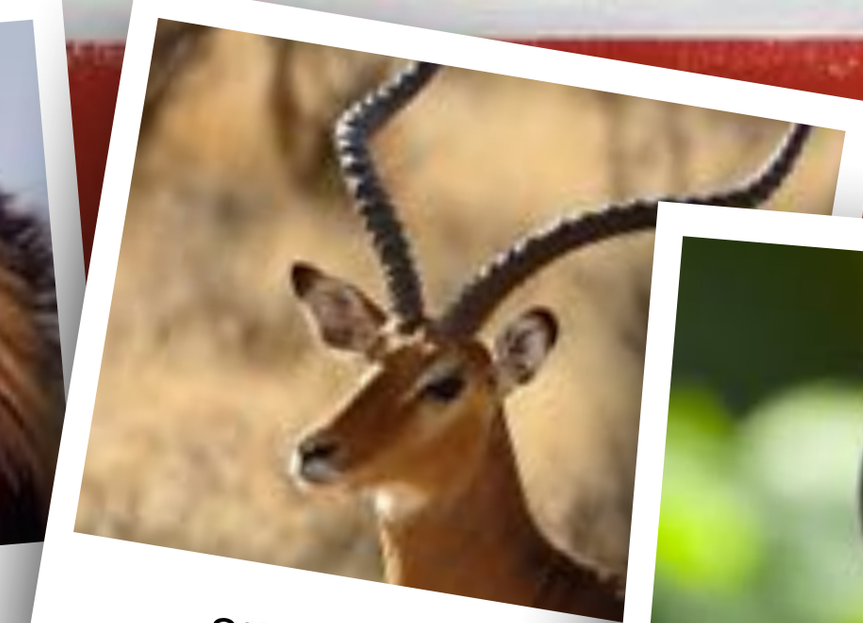
# D OF MORPHISMS

slide from Nicolas Wu


# D OF MORPHISMS

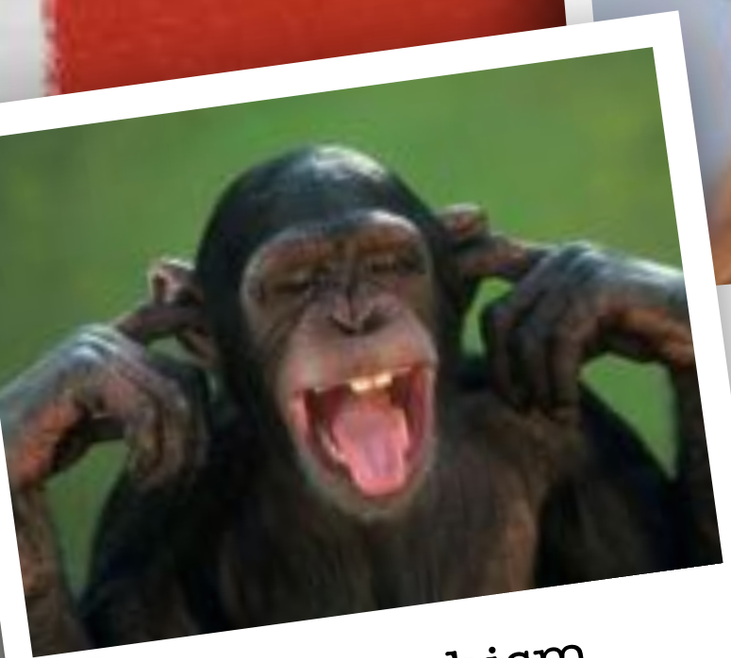
slide from Nicolas Wu

catamorphism



anamorphism

paramorphism



apomorphism

# D OF MORPHISMS

mutumorphism

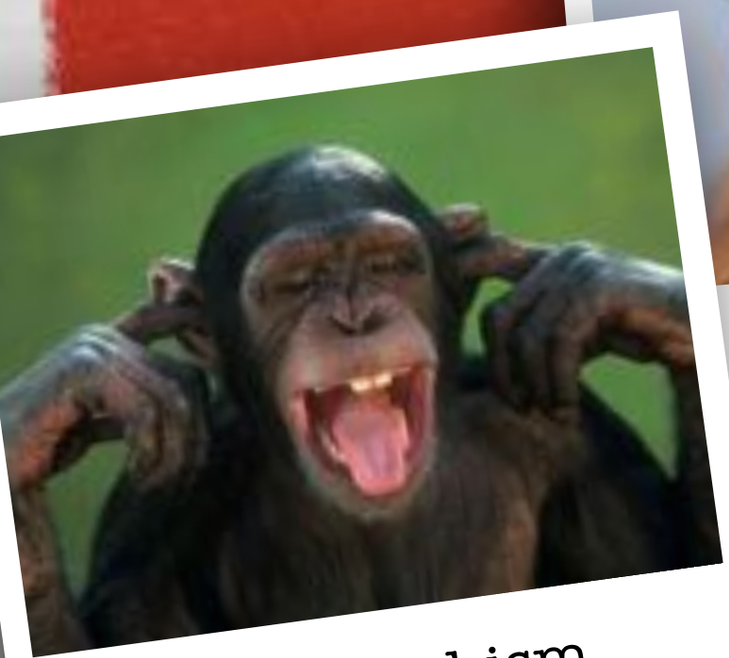
slide from Nicolas Wu

slide from Nicolas Wu

catam hism

anamorphism

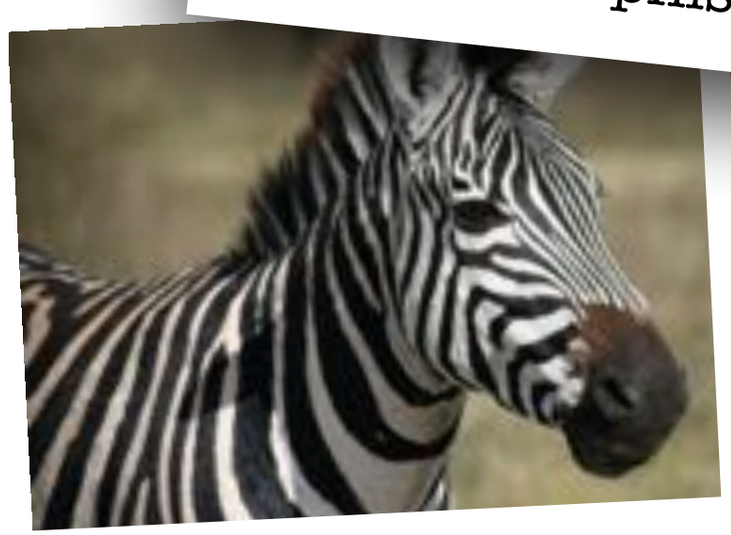
paramorphism



apomorphism

histomorphism

mutumorphism



zygomorphism

O HISMS

slide from Nicolas Wu

slide from Nicolas Wu

slide from Nicolas Wu