

Structured Traversals for (Multiply) Recursive Algebraic Datatypes

G. Cassian Alexandru

January 19, 2021

Presentation generated from .lhs sources using lhs2TeX

Context & Conventions

- Language: Haskell, with numerous language extensions
- Syntactic (e.g. `LambdaCase`)
- Clarifying (e.g. `TypeApplications`, `InstanceSigs`)
- Limited Dependent programming (e.g. `DataKinds`), for multiple recursion

Context & Conventions

- Language: Haskell, with numerous language extensions
- Syntactic (e.g. `LambdaCase`)
- Clarifying (e.g. `TypeApplications`, `InstanceSigs`)
- Limited Dependent programming (e.g. `DataKinds`), for multiple recursion

```
{-# LANGUAGE LambdaCase #-}  
foo :: [a] → b  
foo = λcase  
  [] → ...  
  (x:xs) → ...
```

Context & Conventions

- Language: Haskell, with numerous language extensions
- Syntactic (e.g. `LambdaCase`)
- Clarifying (e.g. `TypeApplications`, `InstanceSigs`)
- Limited Dependent programming (e.g. `DataKinds`), for multiple recursion

```
{-# LANGUAGE LambdaCase #-}  
foo :: [a] → b  
foo = λcase  
  [] → ...  
  (x:xs) → ...
```

- Composition in diagrammatic order: $f;g$ reads “ f , then g ”
- Haskell: $f \mathbin{.}> g$

Structure

1 Single Recursion

- Motivation
- Theory
- Implementation

2 Mutual Recursion

- Motivation
- Theory
- Implementation
- Worked Example

3 Conclusion

```
length :: [a] → Int
```

```
length = λcase
```

```
  [] → 0
```

```
  (_:xs) → 1 + length xs
```

```
filter :: (a → Bool) → [a] → [a]
```

```
filter p = go where
```

```
  go [] = []
```

```
  go (x:xs) = if p x then [x] else [] ++ go xs
```

```
length :: [a] → Int
```

```
length = λcase
```

```
  [] → 0
```

```
  (_:xs) → 1 + length xs
```

```
filter :: (a → Bool) → [a] → [a]
```

```
filter p = go where
```

```
  go [] = []
```

```
  go (x:xs) = if p x then [x] else [] ++ go xs
```

```
length :: [a] → Int
```

```
length = λcase
```

```
  [] → 0
```

```
  (_:xs) → 1 + length xs
```

```
filter :: (a → Bool) → [a] → [a]
```

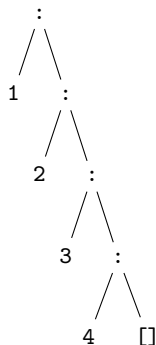
```
filter p = go where
```

```
  go [] = []
```

```
  go (x:xs) = if p x then [x] else [] ++ go xs
```

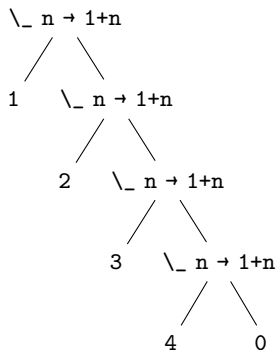
- List Design pattern?
- Design Patterns are a poor man's abstraction
- Recognize common structure & find correct abstract notion

List

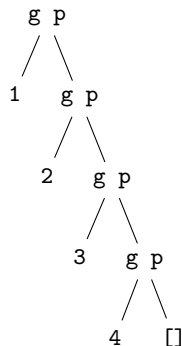


Traversals

length



filter p

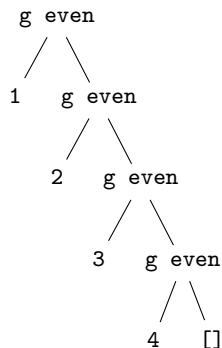


```

g p x xs =
  (if p x then [x] else [])
  ++ xs
  
```

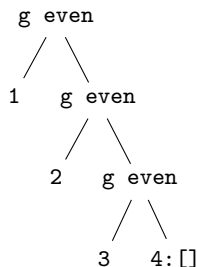
Example Evaluation of filter even

```
g p x xs =  
  (if p x then [x] else []) ++ xs
```



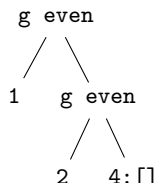
Example Evaluation of filter even

```
g p x xs =  
  (if p x then [x] else []) ++ xs
```



Example Evaluation of filter even

```
g p x xs =  
  (if p x then [x] else []) ++ xs
```



Example Evaluation of filter even

```
g p x xs =  
  (if p x then [x] else []) ++ xs
```

```
g even
```

```
  /  \  
1    2:4: []
```

Example Evaluation of filter even

```
g p x xs =  
  (if p x then [x] else []) ++ xs
```

```
2:4:[]
```

Insight

Even though the functions were defined recursively, their behaviour can be understood non-recursively as simply replacing the two constructors for `[a]` by functions of the same arity.

```
data List a = Nil | Cons a (List a)
```


GADT Syntax:

data *List* a **where**

Nil :: *List* a

Cons :: a → (*List* a) → (*List* a)

GADT Syntax:

```
data List a where
```

```
  Nil :: List a
```

```
  Cons :: a → (List a) → (List a)
```

```
list :: b → (a → b → b) → List a → b
```

```
list nil cons = fold where
```

```
  fold Nil = nil
```

```
  fold (x 'Cons' xs) = x 'cons' fold xs
```

```
length' = list 0 (\_ n → 1+n)
```

```
filter' p = list []
```

```
  (λx xs → (if p x then [x] else [])) ++ xs)
```

GADT Syntax:

data *List* a **where**

Nil :: *List* a

Cons :: a → (*List* a) → (*List* a)

list :: b → (a → b → b) → *List* a → b

list **nil** **cons** = fold **where**

fold *Nil* = nil

fold (x '*Cons*' xs) = x 'cons' fold xs

length' = list 0 (_ n → 1+n)

filter' p = list []

(λx xs → (**if** p x **then** [x] **else** [])) ++ xs)

Now for Expressions

data *Expr* **where**

Lit :: *Int* → *Expr*

Plus :: *Expr* → *Expr* → *Expr*

expr :: (*Int* → b) → (b → b → b) → *Expr* → b

expr *lit* *plus* = fold **where**

fold (*Lit* i) = *lit* i

fold (l '*Plus*' r) = (fold l) 'plus' (fold r)

- We want to define a polytypic “fold”, subsuming `list`, `expr`, which encapsulates the whole “replace constructors with functions” pattern
- We need a deeper understanding of what the datatypes we are working with *are*
- \leadsto Introduce a little ~~Anarchy~~Category Theory

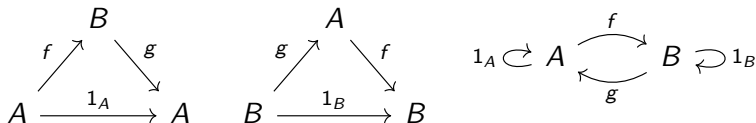
Category

A category \mathcal{C} consists of collections \mathcal{C}_0 of objects and \mathcal{C}_1 of morphisms (or arrows) between them, with the following structure:

- For every two arrows $f : A \rightarrow B$, $g : B \rightarrow C$, there is a composite arrow $f; g : A \rightarrow C$
- For every object $A : \mathcal{C}_0$ there is an identity morphism $1_A : A \rightarrow A$
- Such that the following hold:
 - Composition is associative, that is: $f; (g; h) = (f; g); h$.
 - Composition satisfies unit laws: For every $f : A \rightarrow B$. $id_A; f = f$, $f; id_B = f$.

Isomorphisms

Given a category \mathcal{C} and two objects $A, B : \mathcal{C}_0$, we say A and B are isomorphic via $f : A \rightarrow B$, if there exists a $g : B \rightarrow A$ which is both a left- and right-inverse:



Functors

Let \mathcal{C}, \mathcal{D} be Categories. A *Functor* F is a pair of maps (F_0, F_1) , on the objects and morphisms of the category respectively, such that commuting diagrams are preserved, e.g.:

$$\begin{array}{ccc} & B & \\ f \nearrow & & \searrow g \\ A & \xrightarrow{h} & C \end{array} \xRightarrow{F} \begin{array}{ccc} & F_0 B & \\ F_1 f \nearrow & & \searrow F_1 g \\ F_0 A & \xrightarrow{F_1 h} & F_0 C \end{array}$$

In particular, this means identities & composition are preserved.

Functors

Let \mathcal{C}, \mathcal{D} be Categories. A *Functor* F is a pair of maps (F_0, F_1) , on the objects and morphisms of the category respectively, such that commuting diagrams are preserved, e.g.:

$$\begin{array}{ccc}
 & B & \\
 f \nearrow & & \searrow g \\
 A & \xrightarrow{h} & C
 \end{array}
 \xRightarrow{F}
 \begin{array}{ccc}
 & F_0 B & \\
 F_1 f \nearrow & & \searrow F_1 g \\
 F_0 A & \xrightarrow{F_1 h} & F_0 C
 \end{array}$$

In particular, this means identities & composition are preserved. We will often only write out the definition of a functor on objects.

Functors

Let \mathcal{C}, \mathcal{D} be Categories. A *Functor* F is a pair of maps (F_0, F_1) , on the objects and morphisms of the category respectively, such that commuting diagrams are preserved, e.g.:

$$\begin{array}{ccc}
 & B & \\
 f \nearrow & & \searrow g \\
 A & \xrightarrow{h} & C
 \end{array}
 \xRightarrow{F}
 \begin{array}{ccc}
 & F_0 B & \\
 F_1 f \nearrow & & \searrow F_1 g \\
 F_0 A & \xrightarrow{F_1 h} & F_0 C
 \end{array}$$

In particular, this means identities & composition are preserved. We will often only write out the definition of a functor on objects. When $\mathcal{C} = \mathcal{D}$, we say F is an *Endofunctor*.

Building the Functor kit

- Identity ($IX := X$) is a functor.
- Constant-to- A ($K_A X := A$), for $A : C_0$, is a functor.

Categories can have products (\times_C) and/or coproducts ($+_C$). Think of coproducts as indexed unions in our case. Then if F, G are functors so are:

- $(F \times G)X := FX \times GX$
- $(F + G)X := FX + GX$

Algebra

Let $F : \mathcal{C} \rightarrow \mathcal{C}$ be an endofunctor $A : \mathcal{C}$, $\varphi : FA \rightarrow A$. Then $FA \xrightarrow{\varphi} A$ (or (A, φ)) is an *Algebra*, and A its *Carrier*.

Algebra

$$\begin{array}{c} FA \\ \downarrow \varphi \\ A \end{array}$$

Algebra-Hom:

$$(A, \varphi) \xrightarrow{f} (B, \psi)$$

$$\begin{array}{ccc} FA & \xrightarrow{Ff} & FB \\ \downarrow \varphi & & \downarrow \psi \\ A & \xrightarrow{f} & B \end{array}$$

Initial Algebra: (A, α)

s.t. $\forall (B, \psi)$.

$$\begin{array}{ccc} FA & \xrightarrow{Fh} & FB \\ \downarrow \alpha & & \downarrow \psi \\ A & \xrightarrow{h} & B \end{array}$$

Lambek's Lemma

If F has an initial algebra (A, α) , then A is isomorphic to FA via α . Proof (We show only $h; \alpha = id_A$): Consider the algebra $(FA, F\alpha)$:

$$\begin{array}{c} FA \\ \downarrow \alpha \\ A \end{array}$$

Lambek's Lemma

If F has an initial algebra (A, α) , then A is isomorphic to FA via α . Proof (We show only $h; \alpha = id_A$): Consider the algebra $(FA, F\alpha)$:

$$\begin{array}{ccc}
 FA & \xrightarrow{Fh} & F(FA) \\
 \downarrow \alpha & & \downarrow F\alpha \\
 A & \xrightarrow{h} & FA
 \end{array}$$

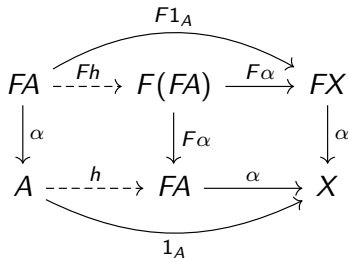
Lambek's Lemma

If F has an initial algebra (A, α) , then A is isomorphic to FA via α . Proof (We show only $h; \alpha = id_A$): Consider the algebra $(FA, F\alpha)$:

$$\begin{array}{ccccc}
 FA & \xrightarrow{Fh} & F(FA) & \xrightarrow{F\alpha} & FX \\
 \downarrow \alpha & & \downarrow F\alpha & & \downarrow \alpha \\
 A & \xrightarrow{h} & FA & \xrightarrow{\alpha} & X
 \end{array}$$

Lambek's Lemma

If F has an initial algebra (A, α) , then A is isomorphic to FA via α . Proof (We show only $h; \alpha = id_A$): Consider the algebra $(FA, F\alpha)$:



Fixed Point

The carrier A of the initial algebra (A, α) of a functor F is a least fixed point of F . Least, that is, in that there is a morphism from it to any other algebra, by initiality.

Fixed Point

The carrier A of the initial algebra (A, α) of a functor F is a least fixed point of F . Least, that is, in that there is a morphism from it to any other algebra, by initiality.

$$\begin{array}{ccc}
 FA & \xrightarrow{Fh} & FB \\
 \alpha^{-1} \uparrow \downarrow \alpha & \circlearrowleft & \downarrow \psi \\
 A & \xrightarrow{h} & B
 \end{array}$$

We get a recursive definition for h : $h = \alpha^{-1}; Fh; \psi$

Back Again

We will now return to the motivating problem and see how what we have just learned is applicable to its solution. In particular, we will see that:

- Our recursive datatypes correspond to fixpoints of associated “structural/base” functors, and are carriers of their initial algebras
- The non-recursive business logic of the traversals, that is, the functions to replace the constructors, correspond to algebras for this functor
- The morphism h we get given (B, ψ) is the polytypic *fold* we were looking for

Functors in Haskell

- Haskell can be seen as a category, where the objects are types and the arrows functions between them.
- Endofunctors in Haskell can be implemented as type constructors $(* \rightarrow *)$
- Definition on arrows $a \rightarrow b$ via typeclass *Functor*, defining function `fmap`
- $F_1(h : A \rightarrow B) : F_0 A \rightarrow F_0 B \quad \sim$
`fmap @F (h :: A → B) :: F A → F B`

Structural Functors

- We obtain the structural functors for our datatypes by factoring the recursion out of their definition, then adding it back in via a fixed-point operator.
- We compare with how this can be done on the value level:

```
type Endo a = a → a
-- Recursive Definition:
fac :: Endo Int
fac n = n * fac (n-1)
-- Fixpoint operator:
fix :: (a → a) → a
fix f = f (fix f)
-- Factoring Fac:
fac' :: Endo Int
fac' = fix g where
  g :: Endo Int → Endo Int
  g f n = n * f (n-1)
```

Recall our list datatype:

```
data List a :: * =
  Nil | Cons a (List a)

-- Fixpoint operator (typelevel)
type Fix :: (* → *) → *
newtype Fix f = In (f (Fix f))
-- Factoring List
type List' a = Fix (ListF a)
data ListF a l =
  NilF | ConsF a l
```

Structural Functors

- We obtain the structural functors for our datatypes by factoring the recursion out of their definition, then adding it back in via a fixed-point operator.
- We compare with how this can be done on the value level:

```
type Endo a = a → a
-- Recursive Definition:
fac :: Endo Int
fac n = n * fac (n-1)
-- Fixpoint operator:
fix :: (a → a) → a
fix f = f (fix f)
-- Factoring Fac:
fac' :: Endo Int
fac' = fix g where
  g :: Endo Int → Endo Int
  g f n = n * f (n-1)
```

Recall our list datatype:

```
data List a :: * =
  Nil | Cons a (List a)

-- Fixpoint operator (typelevel)
type Fix :: (* → *) → *
newtype Fix f = In (f (Fix f))
-- Factoring List
type List' a = Fix (ListF a)
data ListF a l =
  NilF | ConsF a l
```

Business Logic as Algebra

We can store the functions meant to replace the constructors of a type as an algebra (using the transformation $B^A \times B^C \sim B^{A+C}$):

```
type Algebra f a = f a → a
listBL :: b → (a → b → b) → Algebra (ListF a) b
listBL nil cons = λcase
  NilF → nil
  x 'ConsF' b → x 'cons' b
```

Defining a Functor instance

- Do we have to manually define Functor instances?
- Not if working with *polynomial functors*
- Recall the functor kit from theory

```

type ListF' a l = (K () :+: K a :×: I) l
inG :: ListF a l → ListF' a l
inG = λcase
  NilF → InL $ K ()
  a 'ConsF' l → InR (K a :×: I l)
outG :: ListF' a l → ListF a l
outG = λcase
  InL _ → NilF
  InR (K a :×: I l) → a 'ConsF' l
instance Functor (ListF a) where
  fmap f = inG .> fmap f .> outG

```


Generic/Polytypic Programming

- `fmap` is a polytypic function
- Our iso to the generic representation is still boilerplate, though
- Some essentially polytypic functions derivable in Haskell, (e.g. (\equiv) , via **deriving** *Eq*), `fmap` using
 `{-# LANGUAGE DeriveFunctor #-}`
- Whole topic on its own

Cigar!

Recall the recursive definition $h : A \rightarrow B = \alpha^{-1}; Fh; \psi$. To implement it, we only still lack $\alpha^{-1} : A \rightarrow FA$. Recall that a carrier of the initial algebra for functor F is $Fix\ F$. $unFix :: Fix\ f \rightarrow f\ (Fix\ f)$ is easily defined:

```
unFix (In f) = f
```

- Finally, all parts are assembled for our polytypic *fold* function!
- Also called a *catamorphism*, like “cataclysm”: Collapses a structure into a value (which of course can also again be a structure)

```
cata :: Functor f => Algebra f b -> (Fix f) -> b
cata  $\psi$  = unFix .> fmap (cata  $\psi$ ) .>  $\psi$ 
```

Structure

1 Single Recursion

- Motivation
- Theory
- Implementation

2 Mutual Recursion

- Motivation
- Theory
- Implementation
- Worked Example

3 Conclusion

A simple AST

Consider the datatype

```
data Expr = Lit Int | Var Char | Plus Expr Expr |  
          LetIn Decl Expr  
data Decl = Bind Char Expr | Seq Decl Decl
```

Consider an example expression

```
e1 :: Expr  
e1 = LetIn  
    (('x' 'Bind' (Lit 3))  
     'Seq'  
     ('y' 'Bind' (Lit 4))  
    ) (Var 'x') 'Plus' (Var 'y')
```

Mutual Recursion

- *Expr*, *Decl* are a mutually recursive data family. So it is unclear how we should factor out the correct “structural functor”.
- As we did with fixed points, we will take inspiration from how such situations can be handled on the value level.

Mutual Recursion

- *Expr*, *Decl* are a mutually recursive data family. So it is unclear how we should factor out the correct “structural functor”.
- As we did with fixed points, we will take inspiration from how such situations can be handled on the value level.

```
foo = let  
  even =  $\lambda \text{case } 0 \rightarrow \text{True}; n \rightarrow \text{odd } (n-1)$   
  odd =  $\lambda \text{case } 0 \rightarrow \text{False}; n \rightarrow \text{even } (n-1)$   
in even 42
```

Mutual Recursion

- *Expr*, *Decl* are a mutually recursive data family. So it is unclear how we should factor out the correct “structural functor”.
- As we did with fixed points, we will take inspiration from how such situations can be handled on the value level.

```
foo = let
```

```
  even =  $\lambda \text{case } 0 \rightarrow \text{True}; n \rightarrow \text{odd } (n-1)$ 
```

```
  odd =  $\lambda \text{case } 0 \rightarrow \text{False}; n \rightarrow \text{even } (n-1)$ 
```

```
  in even 42
```

- What if the language provides only a singly recursive `let`?
- Use a tupling trick:

```
bar = let (even, odd) = (
```

```
     $\lambda \text{case } 0 \rightarrow \text{True}; n \rightarrow \text{odd } (n-1),$ 
```

```
     $\lambda \text{case } 0 \rightarrow \text{False}; n \rightarrow \text{even } (n-1)$ 
```

```
  )
```

```
  in even 42
```

Product category

For two categories \mathcal{C} and \mathcal{D} , the *product category* $\mathcal{C} \times \mathcal{D}$ is constituted of the following:

- as objects pairs (C, D) , $C : \mathcal{C}_0$, $D : \mathcal{D}_0$
- as morphisms pairs $(f : X \rightarrow Y, g : A \rightarrow B)$, $f : \mathcal{C}_1$, $g : \mathcal{D}_1$, such that composition is defined componentwise

Functors

In Haskell *Functor* represents endofunctors, so we cannot encode the product category. We can write some pseudocode though, for how our structural functor ought to look:

```
data ExprF (e,d) = (  
    Lit Int | Var Char | Plus e e | LetIn d e  
    , Bind Char e | Seq d d  
    )
```

What we need is another way to represent tuples. A preliminary observation is that an n -tuple A^n can be seen as a function from the finite set of cardinality n to A : $n \rightarrow A$.

Remodeling

The functor for our example family has kind $(2 \rightarrow *) \rightarrow (2 \rightarrow *)$. First we should define this kind 2.

We can define an Enum with two accesors for our expr/decl family, and use it as a kind (lifting its constructors to singleton types), using the GHC extension `DataKinds`

Remodeling

```
data Tag = E | D
data family ASTF1 :: (Tag → *) → (Tag → *)
data instance ASTF1 c E =
    Lit1 Int | Var1 Char | Plus1 (c E) (c E) |
    LetIn1 (c D) (c E)
data instance ASTF1 c D =
    Bind1 Char (c E) | Seq1 (c D) (c D)
```

Remodeling

```
data Tag = E | D
data family ASTF1 :: (Tag → *) → (Tag → *)
data instance ASTF1 c E =
    Lit1 Int | Var1 Char | Plus1 (c E) (c E) |
    LetIn1 (c D) (c E)
data instance ASTF1 c D =
    Bind1 Char (c E) | Seq1 (c D) (c D)
```

The type can be read as “given a table of types to be used in the recursive positions of Expr and Decl, respectively, return a table, where at tag E you find the configured Expr type, at D Decl”.

Remodeling

```
data Tag = E | D
data family ASTF1 :: (Tag → *) → (Tag → *)
data instance ASTF1 c E =
    Lit1 Int | Var1 Char | Plus1 (c E) (c E) |
    LetIn1 (c D) (c E)
data instance ASTF1 c D =
    Bind1 Char (c E) | Seq1 (c D) (c D)
```

The two instances define the the component of the structural functor¹ for *Expr* and *Decl* respectively. The tags on the rhs are used to access the components of *c*. If $c \cong (e, d)$, then $c\ E = e$, $c\ D = d$.

¹Actually, for technical reasons you need to write a GADT. I used type family syntax here since I believe it makes it clearer what is going on

IFunctor

We will need a new typeclass, *IFunctor* (standing for *indexed functor*), for functors of the discussed shape. What is the type of morphisms in $(\mathbf{k} \rightarrow *)$? We saw that in the product category, morphisms were of the form (f, g) . In generalizing, we are looking at a collection of maps of the form:

$$\begin{array}{cccc}
 A_1 & A_2 & \dots & A_k \\
 \downarrow f_1 & \downarrow f_2 & & \downarrow f_k \\
 A'_1 & A'_2 & \dots & A'_k
 \end{array}$$

```
class IFunctor (f :: (k → *) → (k → *)) where
  iFmap :: (∀ (i :: k). r i → r' i) → f r ix → f r' ix
```

Fixpoint & Algebra

Nothing very fancy happens here. Both result in a family and are defined pointwise via $ix :: k$.

```
type IFix :: ((k → *) → (k → *)) → (k → *)
```

```
data IFix f (ix :: k) =  
    IIn (f (IFix f) ix)
```

```
iUnFix :: IFix f ix → f (IFix f) ix
```

```
iUnFix (IIn f) = f
```

```
type Algebra f r ix = f r ix → r ix
```

Cata

```
iCata :: ∀ k (f :: (k → *) → (k → *))  
      (r :: (k → *))  
      (ix :: k) · IFunctor f  =>  
      (∀ (i :: k) · Algebra f r i) → (IFix f ix) → r ix  
iCata ψ = iUnFix .> iFmap (iCata ψ) .> ψ
```


Demo!

Structure

1 Single Recursion

- Motivation
- Theory
- Implementation

2 Mutual Recursion

- Motivation
- Theory
- Implementation
- Worked Example

3 Conclusion

Remarks

- There are a wealth of topics in the main paper and related literature not broached in this talk
- A small taste:
 - *unfolds*, producing datastructures, and more schemes
 - fusion laws derived from Category Theory

Conclusion

- using Category Theory, we were able to give a uniform implementation for a whole class of traversals
- We generalized from single to mutual recursion, noting that we didn't need any additional categorical notions
- Implementation details for mutual recursion are quite messy - boilerplate still exists, but ergonomics & reach of recursion schemes have been increasing since their theoretical beginnings.
- If you know anyone who would like to supervise a research internship in this area, I'm looking.